Approaches to Asynchronous Execution across Languages

Tobias Hoffmann ✉
Albert-Ludwigs-Universität Freiburg, Germany

Abstract

Fundamentally there are two kinds of operations in computing, CPU-bound operations and IO-bound operations, the former being operations where the limiting factor to performance is the speed with which calculations can be performed, whereas the latter are operations where performance is limited by the responses of systems external to the CPU. While for both kinds of operations, parallelization is a commonly taken approach to achieve improvements in performance, the way this parallelization is systematically structured and implemented is quite different for the two applications. In the following, I will specifically focus on the approaches provided by a selection of different languages to efficiently handle specifically IO-bound operations.

2012 ACM Subject Classification Theory of computation → Parallel computing models

Keywords and phrases async, parallelization

1 Overview

The primary distinction of IO-bound tasks is that they generally consist of a small amount of computation, while having relatively long wait times. This means that each IO-bound task does not require much CPU time in total, but importantly, whenever an IO-bound task does have to do computation, it is usually important for it to get the necessary CPU time quickly, so that it can dispatch another message and wait for a response from whatever external system it is interacting with.

So to handle a large number of such tasks simultaneously requires quick and efficient switching of tasks to fully utilize the processing resources available. However, ordinary threads as provided by most operating systems generally require too much time and resources to be created and deleted for them to be utilized directly. Therefore, most systems optimized for asynchronous execution of IO-bound tasks utilize a custom system that is abstracted over, or even entirely independent from, the parallelization systems provided by the underlying system architecture.

Another challenge in developing a system for asynchronous computation is the matter of a usable and safe abstract interface for programmers to interact with, due to it being such a fundamental change in how computation is organized inside a program. For this, two different approaches have developed, which are semantically similar, but conceptually and principally distinct.

One, generally aligned with the imperative programming paradigm, is to consider an asynchronous sequence of computation as one whole, which is then routinely interrupted and later resumed whenever it has to wait from some external asynchronous operation. This is usually implemented with bespoke features introduced into the language itself.

The other, generally aligned with the functional programming paradigm, is to instead consider an asynchronous sequence of computation to be made up of a chain of smaller asynchronous operations, resulting in a monadic structure where smaller asynchronous operations are composed into larger asynchronous operations.

While these two approaches look at the problem from different sides, in effect they result in the same semantic model: Bursts of computation delineated by waiting for external responses.

2 Ocaml

While neither the OCaml compiler nor the standard library `Base` provide any support for asynchronous computation, there are external libraries that introduce the functionality into the language. One is `Async`, which comes with Jane Street's `Core` library, another is `Lwt`.

Async

Since the basic ideas of this library have already been covered in the presentation, their explanation will be kept brief here.

Async does not utilize system threads for parallelization, but rather implements it's own system of non-preemptive user-level threads to handle asynchronous operations [5]. While this means that it does not benefit from the potential increase in performance from utilizing multiple cores, it does avoid the overhead that comes with creating and destroying system threads, while still allowing the user to avoid dead-time when waiting for external responses.

Deferred

For representing the potentially pending result of an asynchronous computation, Async provides a high-level monadic datatype `Deferred.t`. With monadic composition, either using `Deferred.bind` directly or some syntactic sugar for it, these asynchronous computations can be chained together.

For asynchronous computations to be executed, control has to be handed over to the Async scheduler using `Scheduler.go`.

Ivar & upon

While `Deferred` provides a high-level way to compose existing asynchronous operations, to implement asynchronous operations from scratch, a low-level interface is provided via `Ivar` and `upon`.

An `Ivar.t` is a variable which represents the actual value behind a Deferred, which can be manually filled at any time, prompting the Deferred to become determined. How and when the Ivar is filled can be arbitrary implemented, allowing for custom asynchronous operations. For this purpose, there also is a function `upon` provided, which allows for scheduling arbitrary non-asynchronous code to be executed once a certain Deferred becomes determined.

As an example [5]:

Listing 1 Delayer example

```
module Delayer = struct
  type t = { delay: Time.Span.t;
             jobs: (unit -> unit) Queue.t;
           }

  let create delay =
    { delay; jobs = Queue.create () }

  let schedule t thunk =
    let ivar = Ivar.create () in
    Queue.enqueue t.jobs (fun () ->
      upon (thunk ()) (fun x -> Ivar.fill ivar x));
    upon (after t.delay) (fun () ->
      let job = Queue.dequeue_exn t.jobs in
      job ());
    Ivar.read ivar
end;;
```

This somewhat artificially conceived utility allows for scheduling some asynchronous operations to be executed in order, but after some predefined delay of time. Here, an Ivar is used to back the Deferred that is returned from the `schedule` function. This Ivar is filled once, first, the artificial delay has elapsed (`upon (after t.delay) (...)`), and then, the actual asynchronous operation that has been scheduled has become determined (`upon (thunk ()) (...)`).

The monadic bind operator used to compose asynchronous operations can very simply be implemented utilizing IVar and `upon` [5]:

Listing 2 Implementation of bind

```
let bind d ~f =
let i = Ivar.create () in
upon d (fun x -> upon (f x) (fun y -> Ivar.fill i y));
Ivar.read i;;
```

3 Rust

While Rust provides as part of it's compiler implementation and it's standard library a framework for encoding asynchronous tasks, it does not provide a runtime to execute these tasks. Therefore, while the core principles of asynchronous execution are defined by Rust itself, the details of how async tasks are executed is defined independently by different libraries [2].

Future

Listing 3 Definition of Future

```
pub trait Future {
    type Output;

    fn poll(self: Pin<&mut Self>,
            cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

Provided as a trait by the Rust standard library, a Future represents an asynchronous computation that either is done, providing a final result value of the computation, or is still pending, needing to make further progress.

To invoke progress to be made on a Future, it has to be polled, which can either result in the Future completing it's computation, returning the final value, or in it suspending it's computation, ensuring by some way for the wakeup function that is passed in via the calling context `cx` to be called once the Future is ready to be polled again for further progress.

Any `struct` implementing Future therefore has to define in it's `poll` both what computation is to be done to further it's progress in computing it's final value, and, if not complete, by what means the wakeup callback is to be called back to signify to the runtime that the Future can be polled again.

When and how any Future is polled is not defined by the Rust compiler or standard library, but instead rather can be handled differently by different external runtimes.

Syntax

To simplify declaration and composition of asynchronous tasks, the Rust compiler, since the 2018 edition, provides some built-in syntax for handling Futures.

Async & Await

Manually constructing the necessary `struct` and implementing `Future` for it for every piece of asynchronous code is tedious, error prone and redundant. Therefore, the Rust compiler provides a way to declare async functions, closures and code blocks using the special keyword `async`, from which during compilation the necessary implementation of `Future` is generated [1].

Similarly, instead of manually dealing with contexts and polling a Future directly, a keyword `await` is provided by the Rust compiler to wait for a Future to conclude and extract it's resulting value [4].

Listing 4 Example for async & await

```
async fn foo() -> u8 { 5 }

fn bar() -> impl Future<Output = u8> {
    async {
        let x: u8 = foo().await;
        x + 5
    }
}
```

The code inside the async function, closure or code block will become the computation that is to be done to further the computation of the Future it compiles to, wherein any `await`

will become a point from which computation will be resumed when the asynchronous task that is being awaited concludes. For this, any data necessary to continue further computation will be stored in the implicitly generated struct that represents our Future.

This combination of `async` and `await` allows us write and compose asynchronous functions, closures and code blocks in a simple and general, runtime-independent manner.

Join

While we can dispatch and wait for a single asynchronous task with `await`, if we want to dispatch multiple asynchronous tasks at once, awaiting each in turn will not suffice, since the execution of the calling async context will be suspended with the first `await`, therefore resulting in the second asynchronous task to be initiated only once the first one has concluded, &c. This means, that the asynchronous tasks will not be dispatched simultaneously, but rather each in sequence, unnecessarily leaving performance that could be gained with an asynchronous execution model on the table.

Listing 5 Sequential await

```
async fn one() -> usize { 1 }
async fn two() -> usize { 2 }

async fn in_sequence() {
    let x = one().await;
    let y = two().await;
    assert_eq!((x, y), (1, 2));
}
```

To remedy this, the Rust standard library provides a macro for joining up multiple Futures, such that they are all polled together when awaited.

Listing 6 Joined await

```
async fn one() -> usize { 1 }
async fn two() -> usize { 2 }

async fn in_sequence() {
    let (x,y) = join!(one(),two()).await;
    assert_eq!((x, y), (1, 2));
}
```

Runtime

The Rust standard library does not provide an async runtime, meaning that while we can readily declare and compose async functions, we can not actually execute any asynchronous code without the use of an external runtime. Neither does the standard library provide asynchronous versions of IO/network/file/&c utilities. Instead, other than the basic interface described above of `Future`, `async` and `await`, all further asynchronous functionality is implemented by external libraries [2]. Of these, Tokio is by far the most popular, and commonly considered the default choice for the majority of use cases [3].

Tokio

The primary runtime provided by Tokio utilizes a fixed sized pool of system-threads, usually aligned with the number of CPU cores available. To each such thread, called a Processor,

asynchronous Tasks are non-preemptively scheduled to be executed [7]. Alternatively, a single-threaded runtime is provided that uses only the currently running thread directly [9].

To minimize necessary synchronization between Processors, instead of having one queue of Tasks from which all Processors take, each Processor has it's own dedicated queue of Tasks. When a Task spawns further Tasks, they are simply pushed onto the queue of the Processor which runs the Task. When a Processor runs out of tasks, it will attempt to steal Tasks from the queues of other Processors. When a Processor's queue becomes unproportionally full, idle Processors are woken up to steal some of it's load. This way, synchronization between Processors is minimized during normal operation under high load, while minimizing the impact on performance of an non-uniform distribution of load across Processors [7].

In parallel to the fixed size pool of threads used for most asynchronous operations, Tokio provides a second pool threads of variable but bound size for blocking or slow operations [3]. This is necessary due to the non-preemptive scheduling of the primary pool of Processors. If a Task running on the primary pool takes too much time without yielding execution, other Tasks ready for execution will have to wait, impacting overall performance. This separation allows for the primary Processor pool to be optimized for fast switching of quickly yielding Tasks, while at the same time being able to cleanly interoperate with blocking or slow operations.

In addition to the runtime, Tokio also provides a suite of asynchronous IO, networking and file-system operations that mirror the synchronous versions provided by the Rust standard library.

Usage

The following implements a basic web server that asynchronously handles incoming connections [8]:

Listing 7 A basic async webserver

```
use tokio::net::{TcpListener, TcpStream};
use std::io;

async fn process(socket: TcpStream) {
    // ...
}

#[tokio::main]
async fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;

    loop {
        let (socket, _) = listener.accept().await?;

        tokio::spawn(
            async move {
            // Process each socket concurrently.
            process(socket).await
        });
    }
}
```

With the asynchronous version of `TcpListener` provided by Tokio, the program waits asynchronously for incoming connections and spawns a new asynchronous task for each

connected client without blocking the main task.

Notably, the `#[tokio::main]` attribute makes it such that when the `main` function is called (i.e. at the start of the program), a Tokio runtime is created and started, without requiring so to be done manually [6].

References

1 async - rust. `https://doc.rust-lang.org/std/keyword.async.html`. (Accessed on 08/03/2022).

2 The async ecosystem - asynchronous programming in rust. `https://rust-lang.github.io/async-book/08_ecosystem/00_chapter.html`. (Accessed on 08/03/2022).

3 Async rust: What is a runtime? here is how tokio works under the hood. `https://kerkour.com/rust-async-await-what-is-a-runtime`. (Accessed on 08/03/2022).

4 Await expressions - the rust reference. `https://doc.rust-lang.org/reference/expressions/await-expr.html`. (Accessed on 08/03/2022).

5 Concurrent programming with async - real world ocaml. `https://dev.realworldocaml.org/concurrent-programming.html`. (Accessed on 08/03/2022).

6 main in tokio - rust. `https://docs.rs/tokio/latest/tokio/attr.main.html`. (Accessed on 08/03/2022).

7 Making the tokio scheduler 10x faster | tokio - an asynchronous rust runtime. `https://tokio.rs/blog/2019-10-scheduler`. (Accessed on 08/03/2022).

8 spawn in tokio - rust. `https://docs.rs/tokio/latest/tokio/fn.spawn.html`. (Accessed on 08/03/2022).

9 tokio::executor - rust. `hhttps://docs.rs/tokio/latest/tokio/runtime/index.html`. (Accessed on 08/03/2022).