Async Benchmarks Index

Mar 22, 2021

I don't understand performance characteristics of "async" programming when applied to typical HTTP based web applications. Let's say we have a CRUD app with a relational database, where a typical request results in N queries to the database and transfers M bytes over the network. How much (orders of magnitude?) faster/slower would an "async" solution be in comparison to a "threaded" solution?

In this *live* post, I am collecting the benchmarks that help to shed the light on this and related questions. Note that I am definitely not the right person to do this work, so, if there is a better resource, I'll gladly just use that instead. Feel free to send pull requests with benchmarks! Every benchmark will be added, but some might go to the rejected section.

I am interested in understanding differences between several execution models, regardless of programming language:

Threads
: Good old POSIX threads, as implemented on modern Linux.

Stackful Coroutines
: M:N threading, which expose the same programming model as threads, but are implemented by multiplexing several user-space coroutines over a single OS-level thread. The most prominent example here is Go

Stackless Coroutines
: In this model, each concurrent computation is represented by a fixed-size state machine which reacts to events. This model often uses `async / await` syntax for describing and composing state machines using standard control

flow constructs.

Threads With Cooperative Scheduling

This is a mostly hypothetical model of OS threads with an additional primitive for directly switching between two threads of the same process. It is not implemented on Linux (see this presentation for some old work towards that). It is implemented on Windows under the “fiber” branding.

I am also interested in Rust’s specific implementation of stackless coroutines

Benchmarks

https://github.com/jimblandy/context-switch

This is a micro benchmark comparing the cost of primitive operations of threads and stackless as implemented in Rust coroutines. Findings:

- Thread creation is order of magnitude slower
- Threads use order of magnitude more RAM.
- IO-related context switches take the same time
- Thread-to-thread context switches (channel sends) take the same time, *if* threads are pinned to one core. This is surprising to me. I’d expect channel send to be significantly more efficient for either stackful or stackless coroutines.
- Thread-to-thread context switches are order of magnitude slower if there’s no pinning
- Threads hit non-memory resource limitations quickly (it’s hard to spawn > 50k threads).

https://github.com/jkarneges/rust-async-bench

Micro benchmark which compares Rust’s implementation of stackless coroutines with a manually coded state machine. Rust's async/await turns out to not be zero-cost, pure overhead is about 4x. The absolute numbers are still low though, and adding even a single syscall of work reduces the difference to only 10%

https://matklad.github.io/2021/03/12/goroutines-are-not-significantly-smaller-than-threads.html

This is a micro benchmark comparing just the memory overhead of threads and stackful coroutines as implemented in Go. Threads are "times", but not "orders of magnitude" larger.

https://calpaterson.com/async-python-is-not-faster.html

Macro benchmark which compares many different Python web frameworks. The conclusion is that `async` is worse for both latency and throughput. Note two important things. *First*, the servers are run behind a reverse proxy (nginx), which drastically changes IO patterns that are observed by the server. *Second*, Python is not the fastest language, so throughput is roughly correlated with the amount of C code in the stack.

There is also a rebuttal post.

Rejected Benchmarks

https://matej.laitl.cz/bench-actix-rocket/

This is a macro benchmark comparing performance of sync and async Rust web servers. This is the kind of benchmark I want to see, and the analysis is exceptionally good. Sadly, a big part of the analysis is fighting with unreleased version of software and working around bugs, so I don't trust that the results are representative.

https://www.techempower.com/benchmarks/

This is a micro benchmark that pretends to be a macro benchmark. The code is overly optimized to fit a very specific task. I don't think the results are easily transferable to real-world applications. At the same time, lack of the analysis and the "macro" scale of the task itself doesn't help with building a mental model for explaining the observed performance.

https://inside.java/2020/08/07/loom-performance

The opposite of a benchmark actually. This post gives a good theoretical overview of why async might lead to performance improvements. Sadly, it drops the ball when it comes to practice:

> millions of user-mode threads instead of the meager thousands the OS can support.

What is the limiting factor for OS threads?