C++ now

Coroutines at Scale

Implementation Choices at Google

Aaron Jacobs

2024

Hello

Overview

- Weird things about Google
- What does "scale" mean?
- The basics of our coroutine library
- Design goals
  - Natural parameter lifetimes
  - Bounded stack usage
  - Cancellation that's hard to get wrong
- Structured and unstructured concurrency

Coroutines at Google

- Still in early days.
- Turned on C++20 for the monorepo in 2023.
- Some experimentation with coroutines before that.
- Billions of QPS of RPCs processed in prod.
- But only a few codebases so far.

Weird things about Google

- Lots of traffic.
- Lots of engineers.

Target use case

- Mostly waiting for things at the millisecond scale.
  - E.g. waiting for I/O, waiting for RPCs.
- Can tolerate some heap allocations.

At Google, everything is an RPC.

Why not synchronous code?

What does "scale" mean?

Scaling in terms of work done

- Do lots of work.
- Don't spend lots of CPU and RAM on it.

Scaling in terms of people

- Tens of thousands of engineers.
- New to C++, new to async programming.
- Different people over time.

We want users to fall into pits of success.

It should be easy to do the right thing, and hard to do the wrong thing.

The library (basics)

```
namespace k3 {
  // ...
}
```

Co<T> is the result of a coroutine that computes a T:

```
// Read the contents of a remote file.
k3:Co<std::string> ReadRemoteFile(
    std::string server, std::string path);
```

You can await a Co<T> to obtain a T:

```
// Print the contents of a remote file.
k3::Co<void> PrintRemoteFile(
    std::string server, std::string path) {
  std::string data = co_await ReadRemoteFile(server, path);
  std::cout << data;
}
```

Control flow

```
co_await Foo()
```

1. Caller calls `Foo`.
2. `Foo` allocates its frame and returns `Co` to the caller.
3. `co_await` suspends the caller.
4. `co_await` resumes `Foo`, handing off caller's handle.
5. [… `Foo` does its work …]
6. `Foo` finishes, resuming the caller.

Design goal: natural parameter lifetimes

Reference params must live until a function returns.

```
void UseInt(const int& x) {
  printf("%d\n", x);
  DoSomething();
  printf("%d\n", x);
}
```

Are either of these uses of x okay?

```
Co<void> UseInt_Async(const int& x) {
  printf("%d\n", x);
  co_await DoSomething();
  printf("%d\n", x);
}
```

Is this alright?

```
co_await UseInt_Async(17);
```

This is probably alright.

```
co_await UseInt_Async(17);
```

1. Create temporary `int` containing 17.
2. Evaluate call expression to obtain `Co`.
3. Evaluate `co_await` expression (does all the work!).
4. Destroy temporary `int`.

Is this alright?

```
Co<void> co = UseInt_Async(17);
printf("I created the Co!\n");
co_await std::move(co);
```

This is probably not alright.

```
Co<void> co = UseInt_Async(17);
printf("I created the Co!\n");
co_await std::move(co);
```

1. Create temporary `int`.
2. Call `UseInt_Async` to obtain `co`.
3. **Destroy temporary `int`.**
4. Resume `UseInt_Async`.

cppcoro

```
cppcoro::task<> usage_example()
{
  // Calling function creates a new task but doesn't start
  // executing the coroutine yet.
  cppcoro::task<int> countTask = count_lines("foo.txt");

  // ...

  // Coroutine is only started when we later co_await the task.
  int lineCount = co_await countTask;

  std::cout << "line count = " << lineCount << std::endl;
}
```

folly

```
folly::coro::Task<int> example() {
  auto t1 = task1();
  auto t2 = task2();
  int result1 = co_await std::move(t1);
  int result2 = co_await std::move(t2);
  co_return result1 + result2;
}
```

We want it to be hard to get this wrong.

The standard doesn't say what to do:

> *Note 3: If a coroutine has a parameter passed by reference, resuming the coroutine after the lifetime of the entity referred to by that parameter has ended is likely to result in undefined behavior.*

The C++ core guidelines say not to use references at all:

> ***CP.53: Parameters to coroutines should not be passed by reference***
>
> *[…]*
>
> ***Enforcement**: Flag all reference parameters to a coroutine.*

"Just avoid references" isn't comprehensive.

- `std::string_view`
- `std::span`
- Pointers

"Just avoid references" isn't comprehensive.

```
struct S {
  const std::string& str;
  std::string_view sv;
};

Co<void> UseStruct(S s);
```

"Just avoid references" is fighting the language.

```
void UseInt(const int& x) {
  printf("%d\n", x);
  DoSomething();
  printf("%d\n", x);
}

Co<void> UseInt_Async(const int& x) {
  printf("%d\n", x);
  co_await DoSomething();
  printf("%d\n", x);
}
```

What about `this` pointers?

```
class C {
 public:
  int Frobnicate();
  Co<int> Frobnicate_Async();
};
```

"Just avoid references" is an efficiency problem.

Design choice: object lifetimes

- All parameters to a K3 coroutine must be valid until it's done.
- References, pointers, views, spans, all reference-like types.
- `this` pointers too!

How do we rescue references?

Two sorts of callers:

```
// A: very common
co_await UseInt_Async(17);

// B: much less common
Co<void> co = UseInt_Async(17);
printf("I created the Co!\n");
co_await std::move(co);
```

Why doesn't it compile?

`Co` is non-moveable, and `co_await` accepts by value.

```
k3::Co<void> UseInt_Async(const int&);

// Works fine
co_await UseInt_Async(17);

// Compiler error
k3::Co<void> co = UseInt_Async(17);
co_await std::move(c);
```

What if you need the other sort of caller?

Coroutine calls can be packaged into a future.

```
k3::Co<void> UseInt_Async(const int&);

// Totally safe
k3::Future<void> future{UseInt_Async, 17};
printf("I created the future!\n");
co_await std::move(future).Run();
```

Futures can be accepted as parameters.

```
// Run all concurrently, finishing once all finish.
k3::Co<void> FanOut(std::vector<k3::Future<void>> futures);

// Run all concurrently, finishing when the first finishes.
template <typename T>
k3::Co<T> Race(std::vector<k3::Future<T>> futures);
```

Design goal: bounded stack usage

Stack usage is a tragedy of the commons.

Stack overflows are really annoying.

Symmetric transfer of control helps.

```
Co<void> Foo();

Co<void> Bar() {
  // Tail call into the body of Foo.
  co_await Foo();
  // Tail call back from the body of Foo.
  [...]
}
```

[expr.await]/5.1.1:

> *Note 1: […] Any number of coroutines can be successively resumed in this fashion […]*

But what about "wake-up" APIs?

```
class Event {
 public:
  // Wait until notified.
  Co<void> Wait();

  // Wake any current or future waiter.
  void Notify();
};
```

After notifying, there are two runnable coroutines.

```
Co<void> WaitAndDoStuff(Event& event) {
  co_await event.Wait();
  // ...
}

Co<void> NotifyAndDoStuff(Event& event) {
  event.Notify();
  // ...
}
```

Possible behavior: resume inline

```
class Event {
  Co<void> Wait();

  // If there is a waiter, it will run inline until it next
  // suspends.
  void Notify();
};
```

1. Coroutine A calls `Notify`.
2. Coroutine B continues until next suspension.
3. Coroutine B returns control to `Notify`.
4. `Notify` returns control to coroutine A.

But…

Design goal: bounded stack usage

Design choice: coroutines on coroutines—not even once.

At all times there is at most one K3 coroutine running on each thread.

Possible behavior: resume on another thread

```
class Event {
  Co<void> Wait();

  // If there is a waiter, it will start running concurrently
  // on another thread. The calling thread continues on.
  void Notify();
};
```

- Thread 1: coroutine A calls `Notify`.
- Thread 2: coroutine B continues on its way.
- Thread 1: `Notify` returns to coroutine A.

But *which* thread?

Design choice: the core library doesn't add parallelism.

Coroutines resume on whatever thread made them ready to resume.

Solution: resume later on same thread

```
class Event {
  Co<void> Wait();

  // If there is a waiter, it will start running on this
  // thread sometime after the calling coroutine suspends.
  void Notify();
};
```

1. Coroutine A calls `Notify`.
2. `Notify` returns control to coroutine A.
3. Coroutine A continues on until it suspends.
4. Coroutine B resumes.

Bonus outcome: no deadlocks from resuming inline.

```
const absl::MutexLock l(&mu_);
if (++count_ == 17) {
  // No deadlock, even if the waiter takes the mutex.
  seventeen_calls_.Notify();
}
```

Why not executors?

(or thread pools, event loops, …)

Why not executors?

Don't offer unnecessary configurability.

Why not executors?

Can use them explicitly if you want them.

```
// Hop to a specific executor. We need to run there because…
co_await Reschedule(my_executor);
```

Why not executors?

"Inline" executors break the programming model.

Design goal: cancellation

Cancellation is important.

- RPCs to stuck machines
- Request hedging
- Avoiding wasted work
- Timeouts

Cancellation is often accidentally ignored.

Design choice: the callee exists only to serve the caller.

It must stop promptly if the caller loses interest.

If a coroutine is stopped early, it won't wake from suspension. Destructors still run.

```
k3::Co<void> SleepAwhile() {
  absl::Cleanup complain_of_poor_sleep = [] {
    printf("Woken early -- caller lost interest.\n");
  };

  // Attempt to get some sleep.
  co_await k3::Sleep(absl::Seconds(2));

  // Nobody interrupted our sleep!
  std::move(complain_of_poor_sleep).Cancel();
}
```

Concurrency

Fork/join concurrency

```
template <typename T>
class EagerFuture {
 public:
  // Start running the supplied future when the current
  // coroutine next suspends.
  explicit EagerFuture(Future<T> future);

  // Wait for it to finish and return its result.
  Co<T> Join();
};
```

Structured concurrency

```
// Run the supplied futures concurrently, finishing once they
// all finish.
Co<void> FanOut(std::vector<Future<void>> futures);
```

Structured concurrency makes object lifetimes easy.

```
// Perfectly safe.
int some_local;
std::string some_other_local;

co_await FanOut({
  Future(&UseInt, std::ref(some_local)),
  Future(&UseString, std::ref(some_other_local)),
});
```

You need to be more careful with unstructured concurrency.

```
std::optional<EagerFuture<void>> worker;

{
  int some_local;
  worker.emplace(Future(&UseInt, std::ref(some_local)));
}  // Whoops
```

Cancellation is obvious with structured concurrency.

```
// If we are cancelled, Foo and Bar will be too. Our
// destructors will run after theirs.
co_await FanOut({
  Future(&Foo),
  Future(&Bar),
});
```

How does that work under the covers?

```
class FanOutAwaitable
  : public internal::AwaitableExpr<void> {
 private:
  internal::LowCo Cancel() {
    for (auto& child : children_) {
      co_await child.Cancel();
    }
  }
};
```

What about unstructured cancellation?

```
// Stop the coroutine if it hasn't yet finished.
//
// By the time this returns, all associated resources have
// been destroyed.
~EagerFuture();
```

Oops, stack overflow!

```
Co<void> Foo(const uint64_t n) {
  std::optional<EagerFuture<void>> ef;
  if (n != 0) {
    ef.emplace(Future(&Foo, n-1));
  }

  co_await NeverResume();
}

Co<void> Bang() {
  EagerFuture foo(Future(&Foo, 1'000'000));
  co_await SleepFor(absl::Seconds(1));
}
```

This time I don't have a clever fix for you.

I lied! Sort of.

```
struct FooState {
  internal::LowCo CleanUp() && {
    if (child.has_value()) {
      co_await child->Stop();
    }
  }

  std::optional<EagerFuture<void>> child;
};

Co<void> FooImpl(internal::AsyncCleanup<FooState> state) {
  if (n != 0) { state->child.emplace(...); }
  co_await NeverResume();
}
```

Can we have async destructors please?

Prefer structured concurrency.

Bounded parallelism is also a problem for unstructured concurrency.

```
Co<absl::Status> MakeRpcUsingThreadPool();

Co<void> Foo() {
  EagerFuture make_rpc(MakeRpcUsingThreadPool);
  co_await DoSomethingElse();
  CHECK_OK(co_await std::move(make_rpc).Join());
}
```

Destructors are great.

Unustructured concurrency is still useful sometimes.

```
class MyClass {
 private:
  const EagerFuture<void> do_background_work_;
};
```

How did we do?

Natural parameter lifetimes: ✅

- **Choice**: object lifetimes work like normal functions.
- **Pit of success**: lifetimes just work within a function.
- **Pit of success**: lifetimes just work in normal calls.
- `Future` does a shallow copy for other uses.

Bounded stack usage: ✅*

- **Design choice**: coroutines never run on top of one another.
- **Pit of success**: stack overflows are impossible.*
- **Pit of success**: no deadlocks when waking.

Cancellation: ✅

- **Design choice**: the callee exists to serve the caller.
- **Pit of success**: impossible to accidentally ignore cancellation.

Lessons

Thoughtful API design can help with efficiency.

Lessons

Good API design *also* helps with usability and safety.

Thank you.

Bonus slides: wrapper functions

Guaranteed copy elision: the cause of and solution to all our problems.

```
template <typename T>
auto await_transform(Co<T> co);
```

Guaranteed copy elision: the cause of and solution to all our problems.

```
Co<void> UseString(const std::string& s);

Co<void> SomeWrapper(const char* const s) {
  return UseString(s);  // Whoops
}
```

Guaranteed copy elision: the cause of and solution to all our problems.

```
std::function<Co<void>(const char*)> f = UseString;
co_await f("taco");  // Whoops
```

Static analysis: the true solution to all of our problems.

```
template <typename T>
class [[clang::coro_return_type]] Co { … };

// Doesn't even compile. Thanks Utkarsh!
Co<void> SomeWrapper(const char* const s) {
  return UseString(s);
}
```

Can still affirm you know what you're doing.

```
template <typename... Args>
Co<void> UseReferences(Args&&... args);

template <typename... Args>
[[clang::coro_wrapper]]
Co<void> SafeWrapper(Args&&... args) {
  printf("hi\n");
  return UseReferences(std::forward<Args>(args)...);
}
```

There's always more work to do.

```
// Doesn't compile. :-/
std::function<Co<void>(const char*)> f = UseString;
```

Thank you.