Document number: P2073R0
Date: 2020-01-13 (pre-Prague mailing)
Reply To: Dmitry Duka (dduka@nvidia.com)
Ivan Shutov (ishutov@nvidia.com)
Konstantin Sadov (ksadov@nvidia.com)
Contributors: Marco Foco (mfoco@nvidia.com)
Audience: WG21, Tooling Study Group SG15

Debugging C++ coroutines

Introduction

Code samples

- Integration code
- Usage example

Windows

Linux

Introduction

The following is a report summarizing our evaluation of C++ coroutines as proposed in P0664R7. We've ported existing code implementing a websocket server based on callbacks to coroutines. The prototype of the server that we've got passes tests on both Windows and Linux. However we've found major downsides to using coroutines while debugging them. Ability to check values of input arguments and local variables of a coroutine in a debugger is either missing or at the very least cumbersome, depending on the platform. Since coroutines in the form that we're going to have them in C++20 in most useful cases require user code integrating the language with a particular user-defined execution context, ability to check promise_type object of a coroutine is required for debugging. Such ability is either missing or too cumbersome for everyday use, depending on the platform.

Code samples

Before we discuss our findings on both platforms, we would like to illustrate how we've integrated C++ coroutines. The short summary is that we wanted to integrate coroutines such that it would be possible to choose different types of execution contexts - thread pool, single worker thread, fiber based task manager, etc. The coroutine body is always executed inside execution context thread across all its suspend points. The code below doesn't handle exceptions and doesn't cover all edge cases, move semantics, etc, but should be enough to outline the summary of our findings.

Integration code

```
// Abstract interface to some execution context.
// It can be simply a separate thread awaiting for handles
// and calling .resume() member function for each handle once.
// If a coroutine has multiple suspend points, it’s
// responsibility of the Scheduler interface to put a
// coroutine handle in the Execution context multiple times.
class ExecutionContext
{
public:
    virtual void add(coroutine_handle<> handle) = 0;
};

// Awaiter interface for resuming parent coroutines
struct run_parent_awaiter
{
    bool await_ready()
    {
        return false;
    }
    template<typename promise>
    void await_suspend(coroutine_handle<promise> handle)
    {
        auto clbk = handle.promise().mFinalCallback;
        if (clbk)
        {
            clbk(handle, handle.promise().mFinalUserPtr);
        }
        if (handle.promise().mParentHandle)
        {
            handle.promise().mExecutionContext->add(handle.promise().mParentHandle);
        }
        else
        {
            // It doesn't have parent, it is top level coro
            handle.destroy();
        }
    }
    void await_resume() {}
};

// Base class for all promise_type types
struct promise_base
{
    typedef void(*FinalCallback)(coroutine_handle<> caller, void *userPtr);
    ExecutionContext* mExecutionContext;
    coroutine_handle<> mParentHandle;

    FinalCallback mFinalCallback;
    void *mFinalUserPtr;

    promise_base() : mExecutionContext(nullptr), mParentHandle(), mFinalCallback(nullptr),
mFinalUserPtr(nullptr) {}

    suspend_always initial_suspend()
    {
        return {};
    }
```

```
    void setFinalCallback(FinalCallback callback, void *userPtr)
    {
        mFinalCallback = callback;
        mFinalUserPtr = userPtr;
    }

    void unhandled_exception() {}
};

// Awaiter interface
template<typename ResultType> struct Coro
{
    struct promise_type : public promise_base
    {
        ResultType result;
        std::unique_ptr<std::promise<ResultType>> promise;

        void return_value(ResultType& v)
        {
            result = v;
            if (promise)
            {
                promise->set_value(result);
            }
        }

        auto final_suspend()
        {
            return run_parent_awaiter{};
        }

        Coro<ResultType> get_return_object();
    };

    typedef coroutine_handle<promise_type> HandleType;
    HandleType mHandle;

    Coro(HandleType handle) : mHandle(handle) {}

    ~Coro()
    {
        if (mHandle.promise().mParentHandle)
        {
            mHandle.destroy();
        }
    }

    CoroAwaiter<ResultType, promise_type> operator co_await()
    {
        return { mHandle };
    }

    std::future<ResultType> runOn(ExecutionContext* exec)
    {
        mHandle.promise().promise.reset(new std::promise<ResultType>());
        mHandle.promise().mExecutionContext = exec;
        exec->add(mHandle);
        return mHandle.promise().promise->get_future();
    }
};
```

Usage example

```
Coro<uint64_t> add(uint64_t x, uint64_t y)
{
    co_await remoteLog(...);
    co_return x + y;
}
```

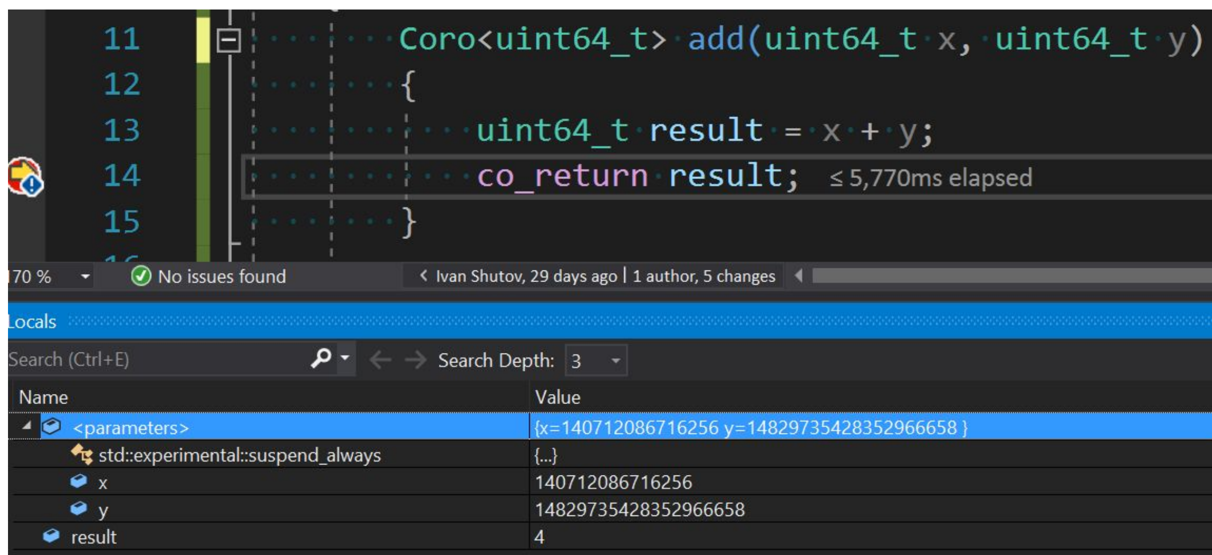
Windows

On Windows we've evaluated the following tools:

- v141u5, 19.12.25831.0, Visual Studio 2017 Professional Update 5, version 15.5.2
- v142, 19.24.28314.0, Visual Studio Professional 2019, version 16.4.2

We used debuggers supplied with both versions of Visual Studio.

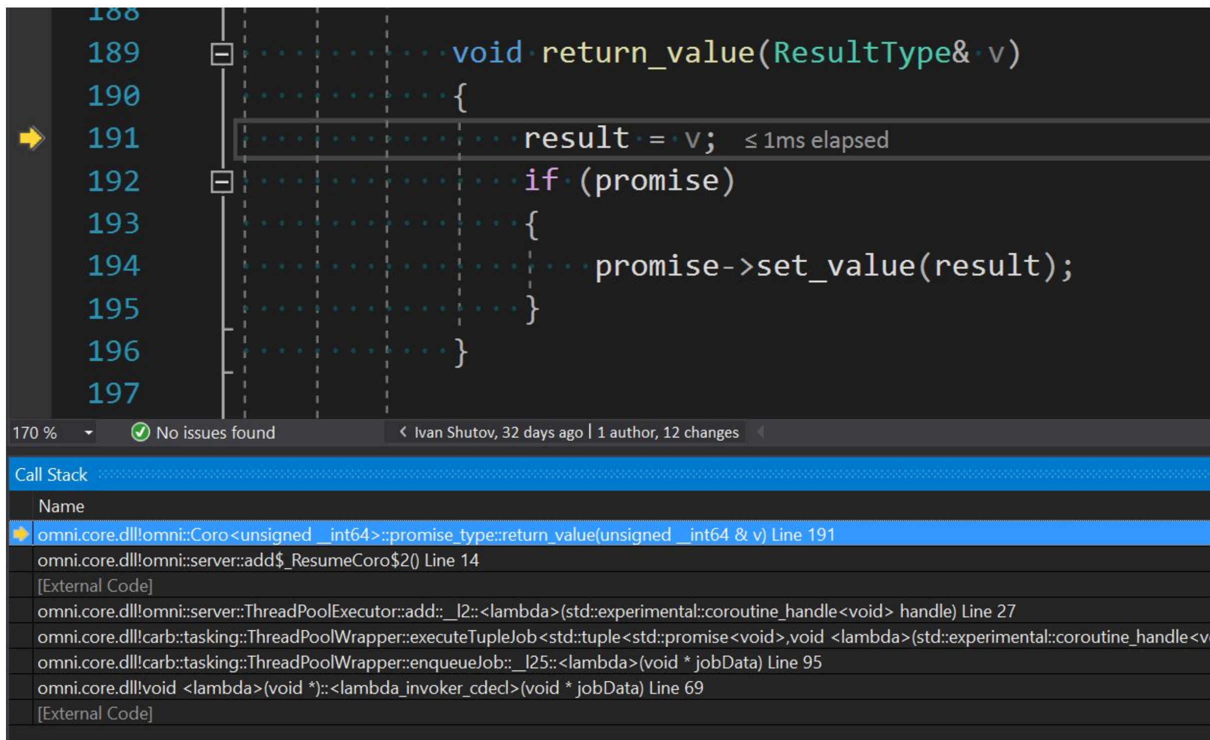
Visual Studio 2019 debugger, v141u5, 19.12.25831.0 toolchain:



A couple of things should be noted:

- **x** and **y** arguments are not resolved by the debugger
- **result** is resolved
- It's not possible to get to the promise_type object of the currently running coroutine

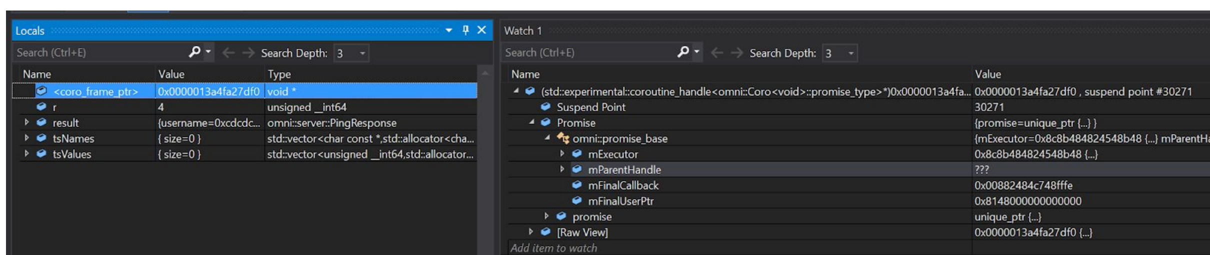
The last issue can be mitigated by stepping inside co_return expression:



Although this is not ideal, it makes it possible to debug coroutine integration code. In some cases this is not enough though, as it might be helpful to see the state of promise_type object without having to advance the program to the nearest Coroutine TS customization point which would trigger a call to one of the functions defined in the integration code.

Visual Studio 2019 debugger, v142, 19.24.28314.0 toolchain:

Newer toolchain yields slightly better results, although still far from ideal.



Notice **<coro_frame_ptr>** is now visible within the coroutine frame. It is then possible to cast the pointer manually to the coroutine_handle and resolve the promise_type of the currently running coroutine. It is still not possible to get to the values of input arguments.

One solution for this problem is to convert input arguments to local variables by using some sort of macro, a capturing function or just copy arguments to local variables manually, just for debugging purposes, perhaps only in debug builds. Again, this is suboptimal, but unblocks debugging somewhat.

Linux

On Linux we've evaluated the following toolchains:

- clang-7.0.0
- clang-9.0.0

We used gdb (v8.1.0) and lldb (v7.0.0 and v9.0.0) debuggers

lldb v9.0.0, clang-9.0.0 toolchain:

We evaluate how a simple coroutine with a suspend point in the middle of it behaves under debugger:

```
   15
   16          Coro<float> test(float a, float b)
   17          {
-> 18              float c = a * b;
   19              float d = co_await otherCoro();
   20              co_return a + b + c + d;
   21          }
(lldb) fr v
(std::experimental::coroutines_v1::__coroutine_traits_sfinae<omni::Coro<float> >::promise_type) __pro
mise = <variable not available>

(float) c = <variable not available>

(float) d = <variable not available>

(lldb) print a
error: Couldn't materialize: couldn't get the value of variable a: variable not available
error: errored out in DoExecute, couldn't PrepareToExecuteJITExpression
```

Notice, value of input arguments **a** and **b** are not accessible. It is possible to access **c** though:

```
p:19:32
   16          Coro<float> test(float a, float b)
   17          {
   18              float c = a * b;
-> 19              float d = co_await otherCoro();
   20              co_return a + b + c + d;
   21          }
   22
(lldb) fr v
(std::experimental::coroutines_v1::__coroutine_traits_sfinae<omni::Coro<float> >::promise_type) __pro
mise = <variable not available>

(float) c = 50
(float) d = <variable not available>
```

But, only until the next suspend point of the coroutine:

```
   17           {
   18               float c = a * b;
   19               float d = co_await otherCoro();
-> 20               co_return a + b + c + d;
   21           }
   22
   23           omni::Coro<void> ping(RequestServiceData& rsd, std::vector<std::pair<std::string, TimestampMi
croseconds> > ts, PingCallback clbk, void* userPtr, TimestampMicroseconds inTimestamp)
(lldb) fr v
(std::experimental::coroutines_v1::__coroutine_traits_sfinae<omni::Coro<float> >::promise_type) __promise = <
variable not available>

(float) c = <variable not available>

(float) d = 5
(lldb) print a
error: Couldn't materialize: couldn't get the value of variable a: variable not available
error: errored out in DoExecute, couldn't PrepareToExecuteJITExpression
(lldb) print b
error: Couldn't materialize: couldn't get the value of variable b: variable not available
error: errored out in DoExecute, couldn't PrepareToExecuteJITExpression
```

The situation is very similar in gdb:

```
18                      float c = a * b;
(gdb) info locals
c = <optimized out>
d = <optimized out>
a = <optimized out>
b = <optimized out>
__promise = <optimized out>
```

```
20                      co_return a + b + c + d;
(gdb) info local
c = <optimized out>
d = 5
a = <optimized out>
b = <optimized out>
__promise = <optimized out>
```

Notice, this is debug build, so nothing should have been optimized out. Likely gdb just couldn't resolve symbols because coroutine bytecode behaves differently compared to regular functions.

Conclusion

We conclude this report with the summary table.

	Linux: gdb / lldb + clang 6 / clang 9	Visual Studio debugger + v141u5/v142
set breakpoint	possible	possible
local variables	not visible	visible
input arguments	not visible	not visible
see promise_type object of the currently running coroutine	not visible	only in v142 with some manual steps / casting in the debugger

It seems that toolchains don’t generate code for coroutines which are recognized correctly by debuggers. This may or may not be related to the fact that promise_type objects, input arguments and local variables are stored in the coroutine state and are not located on the stack as expected by debuggers. It seems that either toolchains need to generate code that is more debugger-friendly or debuggers need to handle coroutines in a special way.

It is possible to mitigate issues on Windows, although it doesn’t look debug-friendly at the moment. On Linux the situation is worse. It is not possible to debug a coroutine without looking into disassembly in general case. We believe tooling needs to be ready for C++20 coroutines and provide at least these basic capabilities.