

Modular



Efficient Coroutine Implementation in MLIR

Agenda

-
- 01 Asynchronous Programming Background
 - 02 Coroutine Implementations in LLVM
 - 03 MLIR Coroutine Implementation
-

M



01

Asynchronous Programming

Synchronous Kernel Computation

```
int main() {  
    while(true) {  
        int x1, x2;  
        std::cout << "Enter two integers: ";  
        std::cin >> x1 >> x2;  
        if (x1 < 0 || x2 < 0)  
            break;  
        float *result = kernelCalculation(x1, x2);  
        printf("result[0]: %f\n", result[0]);  
        free(result);  
  
        // simulate other work  
        std::cout << "tend to other things\n";  
        std::this_thread::sleep_for(  
            std::chrono::milliseconds(25));  
  
    }  
}
```

```
float* kernelCalculation(int x1, int x2){  
    Event* A = launchKernelA(x1);  
    Event* B = launchKernelB(x2);  
    eventA->synchronize();  
    eventB->synchronize();  
    Event* C = launchKernelC(A->getData(),B->getData());  
    C->synchronize();  
    C->getData()->cleanup();  
    return C->getData()->hostResult;  
}
```


Synchronous Kernel Computation

```
int main() {  
    while(true) {  
        int x1, x2;  
        std::cout << "Enter two integers: ";  
        std::cin >> x1 >> x2;  
        if (x1 < 0 || x2 < 0)  
            break;  
  
        float *result = kernelCalculation(x1, x2);  
        printf("result[0]: %f\n", result[0]);  
        free(result);  
  
        // simulate other work  
        std::cout << "tend to other things\n";  
        std::this_thread::sleep_for(  
            std::chrono::milliseconds(25));  
  
    }  
}
```

```
float* kernelCalculation(int x1, int x2){  
    Event* A = launchKernelA(x1);  
    Event* B = launchKernelB(x2);  
    eventA->synchronize();  
    eventB->synchronize();  
    Event* C = launchKernelC(A->getData(),B->getData());  
    C->synchronize();  
    C->getData()->cleanup();  
    return C->getData()->hostResult;  
}
```

gather user
input

Synchronous Kernel Computation

```
int main() {  
    while(true) {  
        int x1, x2;  
        std::cout << "Enter two integers: ";  
        std::cin >> x1 >> x2;  
        if (x1 < 0 || x2 < 0)  
            break;  
  
        float *result = kernelCalculation(x1, x2);  
        printf("result[0]: %f\n", result[0]);  
        free(result);  
  
        // simulate other work  
        std::cout << "tend to other things\n";  
        std::this_thread::sleep_for(  
            std::chrono::milliseconds(25));  
  
    }  
}
```

```
float* kernelCalculation(int x1, int x2){  
    Event* A = launchKernelA(x1);  
    Event* B = launchKernelB(x2);  
    eventA->synchronize();  
    eventB->synchronize();  
    Event* C = launchKernelC(A->getData(), B->getData());  
    C->synchronize();  
    C->getData()->cleanup();  
    return C->getData()->hostResult;  
}
```

gather user
input

Write to GPU
command
buffers

Synchronous Kernel Computation

```
int main() {  
    while(true) {  
        int x1, x2;  
        std::cout << "Enter two integers: ";  
        std::cin >> x1 >> x2;  
        if (x1 < 0 || x2 < 0)  
            break;  
  
        float *result = kernelCalculation(x1, x2);  
        printf("result[0]: %f\n", result[0]);  
        free(result);  
  
        // simulate other work  
        std::cout << "tend to other things\n";  
        std::this_thread::sleep_for(  
            std::chrono::milliseconds(25));  
  
    }  
}
```

```
float* kernelCalculation(int x1, int x2){  
    Event* A = launchKernelA(x1);  
    Event* B = launchKernelB(x2);  
  
    eventA->synchronize();  
    eventB->synchronize();  
  
    Event* C = launchKernelC(A->getData(),B->getData());  
    C->synchronize();  
    C->getData()->cleanup();  
    return C->getData()->hostResult;  
}
```

gather user
input

Write to GPU
command
buffers

check status
registers $\triangle$

Synchronous Kernel Computation

<pre>int main() { while(true) { int x1, x2; std::cout << "Enter two integers: "; std::cin >> x1 >> x2; if (x1 < 0 x2 < 0) break; float *result = kernelCalculation(x1, x2); printf("result[0]: %f\n", result[0]); free(result); // simulate other work std::cout << "tend to other things\n"; std::this_thread::sleep_for(std::chrono::milliseconds(25)); } }</pre>		gather user input
		Write to GPU command buffers
<pre>float* kernelCalculation(int x1, int x2){ Event* A = launchKernelA(x1); Event* B = launchKernelB(x2); eventA->synchronize(); eventB->synchronize(); Event* C = launchKernelC(A->getData(),B->getData()); C->synchronize(), C->getData()->cleanup(); return C->getData()->hostResult; }</pre>		check status registers ⚠
		Write to GPU command buffers

Synchronous Kernel Computation

<pre>int main() { while(true) { int x1, x2; std::cout << "Enter two integers: "; std::cin >> x1 >> x2; if (x1 < 0 x2 < 0) break; float *result = kernelCalculation(x1, x2); printf("result[0]: %f\n", result[0]); free(result); // simulate other work std::cout << "tend to other things\n"; std::this_thread::sleep_for(std::chrono::milliseconds(25)); } }</pre>	<pre>float* kernelCalculation(int x1, int x2){ Event* A = launchKernelA(x1); Event* B = launchKernelB(x2); eventA->synchronize(); eventB->synchronize(); Event* C = launchKernelC(A->getData(), B->getData()); C->synchronize(); C->getData()->cleanup(); return C->getData()->hostResult; }</pre>	gather user input
		Write to GPU command buffers
		check status registers ⚡
		Write to GPU command buffers
		check status registers ⚡

Synchronous Kernel Computation

<pre>int main() { while(true) { int x1, x2; std::cout << "Enter two integers: "; std::cin >> x1 >> x2; if (x1 < 0 x2 < 0) break; float *result = kernelCalculation(x1, x2); printf("result[0]: %f\n", result[0]); free(result); // simulate other work std::cout << "tend to other things\n"; std::this_thread::sleep_for(std::chrono::milliseconds(25)); } }</pre>	<pre>float* kernelCalculation(int x1, int x2){ Event* A = launchKernelA(x1); Event* B = launchKernelB(x2); eventA->synchronize(); eventB->synchronize(); Event* C = launchKernelC(A->getData(),B->getData()); C->synchronize(); C->getData()->cleanup(); return C->getData()->hostResult; }</pre>	gather user input
		Write to GPU command buffers
		check status registers ⚠
		Write to GPU command buffers
		check status registers ⚠
		tend to other work

Synchronous Kernel Computation

<pre>int main() { while(true) { int x1, x2; std::cout << "Enter two integers: "; std::cin >> x1 >> x2; if (x1 < 0 x2 < 0) break; float *result = kernelCalculation(x1, x2); printf("result[0]: %f\n", result[0]); free(result); // simulate other work std::cout << "tend to other things\n"; std::this_thread::sleep_for(std::chrono::milliseconds(25)); } }</pre>		gather user input
		Write to GPU command buffers
		check status registers ⚠
		Write to GPU command buffers
		check status registers ⚠
		tend to other work

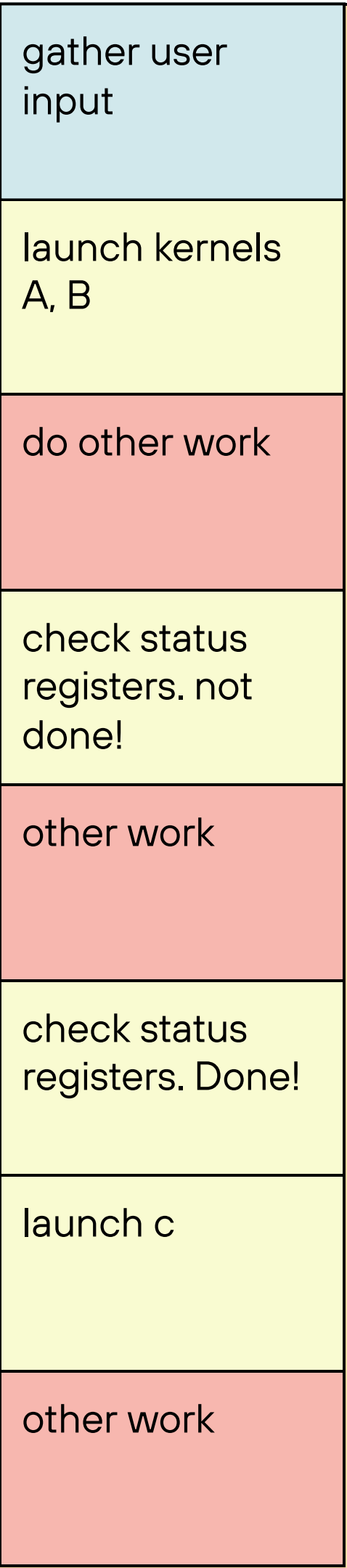
float* kernelCalculation(int x1, int x2){
 Event* A = launchKernelA(x1);
 Event* B = launchKernelB(x2);
 eventA->synchronize();
 eventB->synchronize();
 Event* C = launchKernelC(A->getData(),B->getData());
 C->synchronize();
 C->getData()->cleanup();
 return C->getData()->hostResult;
}

We don't want to block the main application while we check the status registers


```
int main() {
  while(true){
    int x1, x2;
    std::cout << "Enter two integers: ";
    std::cin >> x1 >> x2;
    if(x1 < 0 || x2 < 0)
      break;
    Coro* coro = new Coro(x1, x2);
    kernelCalculation(coro);
    State prevstate = NotStarted;
    while(coro->state != CDone){
      if(coro->state == AandBDone
        && prevstate != AandBDone)
        printf("finished A and B\n");
        prevstate = coro->state;

      // simulate other work
      std::cout << "tend to other things\n";
      std::this_thread::sleep_for(
        std::chrono::milliseconds(25));

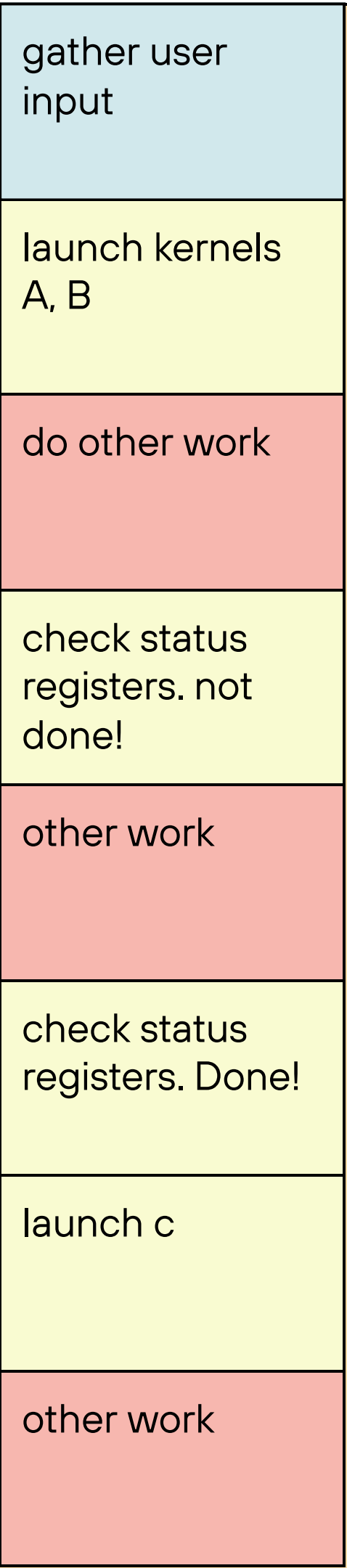
      // check on kernel
      kernelCalculation(coro);
    }
    Data* results = coro->result;
    results->cleanup();
    printf("result[0]: %f\n", results->hostResult[0]);
    free(results->hostResult);
    delete coro;
  }
}
```



```
int main() {
    while(true){
        int x1, x2;
        std::cout << "Enter two integers: ";
        std::cin >> x1 >> x2;
        if(x1 < 0 || x2 < 0)
            break;
        Coro* coro = new Coro(x1, x2);
        kernelCalculation(coro);
        State prevstate = NotStarted;
        while(coro->state != CDone){
            if(coro->state == AandBDone
                && prevstate != AandBDone)
                printf("finished A and B\n");
            prevstate = coro->state;

            // simulate other work
            std::cout << "tend to other things\n";
            std::this_thread::sleep_for(
                std::chrono::milliseconds(25));

            // check on kernel
            kernelCalculation(coro);
        }
        Data* results = coro->result;
        results->cleanup();
        printf("result[0]: %f\n", results->hostResult[0]);
        free(results->hostResult);
        delete coro;
    }
}
```



```
enum State {
    NotStarted = 0,
    Started = 1,
    AandBDone = 2,
    CDone = 3
};

void kernelCalculation(Coro *coro) {
    switch (coro->state){
        case State::NotStarted:
            coro->eventA = launchKernelA(coro->input1);
            coro->eventB = launchKernelB(coro->input2);
            coro->state = State::Started;
            return;
        case State::Started:
            if(coro->eventA->isDone() && coro->eventB->isDone()){
                coro->eventC = launchKernelC(coro->eventA->getData(),
                                                coro->eventB->getData());

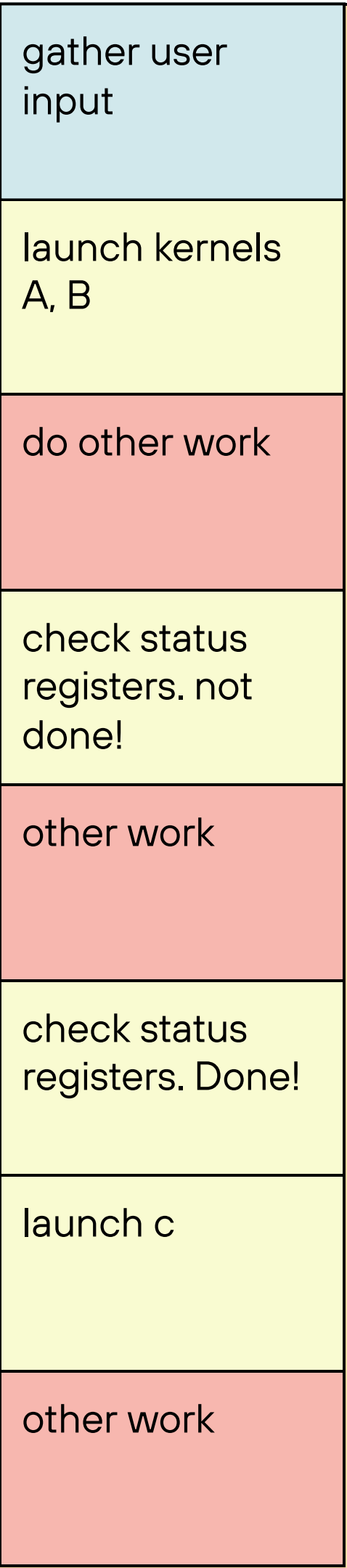
                coro->state = State::AandBDone;
            }
            return;
        case State::AandBDone:
            if(coro->eventC->isDone()){
                coro->result = coro->eventC->getData();
                coro->state = State::CDone;
            }
            return;
        case State::CDone:
            return;
    }
}
```



```
int main() {
    while(true){
        int x1, x2;
        std::cout << "Enter two integers: ";
        std::cin >> x1 >> x2;
        if(x1 < 0 || x2 < 0)
            break;
        Coro* coro = new Coro(x1, x2);
        kernelCalculation(coro);
        State prevstate = NotStarted;
        while(coro->state != CDone){
            if(coro->state == AandBDone
                && prevstate != AandBDone)
                printf("finished A and B\n");
            prevstate = coro->state;

            // simulate other work
            std::cout << "tend to other things\n";
            std::this_thread::sleep_for(
                std::chrono::milliseconds(25));

            // check on kernel
            kernelCalculation(coro);
        }
        Data* results = coro->result;
        results->cleanup();
        printf("result[0]: %f\n", results->hostResult[0]);
        free(results->hostResult);
        delete coro;
    }
}
```



```
enum State {
    NotStarted = 0,
    Started = 1,
    AandBDone = 2,
    CDone = 3
};

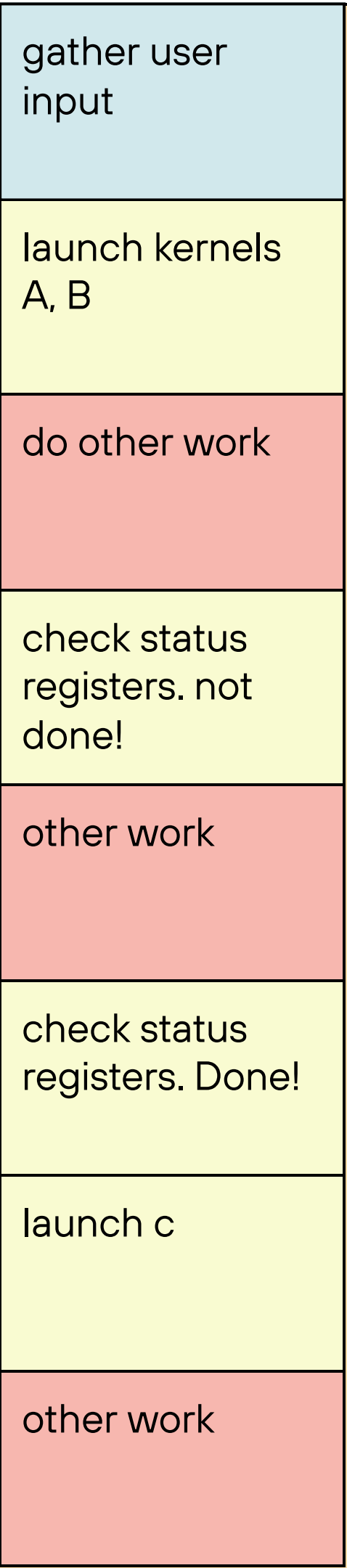
void kernelCalculation(Coro *coro) {
    switch (coro->state){
        case State::NotStarted:
            coro->eventA = launchKernelA(coro->input1);
            coro->eventB = launchKernelB(coro->input2);
            coro->state = State::Started;
            return;
        case State::Started:
            if(coro->eventA->isDone() && coro->eventB->isDone()){
                coro->eventC = launchKernelC(coro->eventA->getData(),
                    coro->eventB->getData());

                coro->state = State::AandBDone;
            }
            return;
        case State::AandBDone:
            if(coro->eventC->isDone()){
                coro->result = coro->eventC->getData();
                coro->state = State::CDone;
            }
            return;
        case State::CDone:
            return;
    }
}
```

```
int main() {
    while(true){
        int x1, x2;
        std::cout << "Enter two integers: ";
        std::cin >> x1 >> x2;
        if(x1 < 0 || x2 < 0)
            break;
        Coro* coro = new Coro(x1, x2);
        kernelCalculation(coro);
        State prevState = NotStarted;
        while(coro->state != CDone){
            if(coro->state == AandBDone
                && prevState != AandBDone)
                printf("finished A and B\n");
            prevState = coro->state;

            // simulate other work
            std::cout << "tend to other things\n";
            std::this_thread::sleep_for(
                std::chrono::milliseconds(25));

            // check on kernel
            kernelCalculation(coro);
        }
        Data* results = coro->result;
        results->cleanup();
        printf("result[0]: %f\n", results->hostResult[0]);
        free(results->hostResult);
        delete coro;
    }
}
```



```
enum State {
    NotStarted = 0,
    Started = 1,
    AandBDone = 2,
    CDone = 3
};

void kernelCalculation(Coro* coro) {
    switch (coro->state){
        case State::NotStarted:
            coro->eventA = launchKernelA(coro);
            coro->eventB = launchKernelB(coro);
            coro->state = State::Started;
            return;
        case State::Started:
            if(coro->eventA->isDone())
                coro->eventC = launchKernelC(coro);
            coro->eventB->getData(),
            coro->state = State::AandBDone;
    }
    return;
    case State::AandBDone:
        if(coro->eventC->isDone()){
            coro->result = coro->eventC->getData();
            coro->state = State::CDone;
        }
        return;
    case State::CDone:
        return;
    }
}
```

```
struct Coro {
    Coro(int x1, int x2): input1(x1), input2(x2),
        state(NotStarted){}
    ~Coro(){
        delete eventA;
        delete eventB;
        delete eventC;
        delete result;
    }
    int input1;
    int input2;
    Event* eventA;
    Event* eventB;
    Event* eventC;
    Data *result;
    State state;
};
```



```
int main() {
    while(true){
        int x1, x2;
        std::cout << "Enter two integers: ";
        std::cin >> x1 >> x2;
        if(x1 < 0 || x2 < 0)
            break;
        Coro* coro = new Coro(x1, x2);
        kernelCalculation(coro);
        State prevState = NotStarted;
        while(coro->state != CDone){
            if(coro->state == AandBDone
                && prevState != AandBDone)
                printf("finished A and B\n");
            prevState = coro->state;

            // simulate other work
            std::cout << "tend to other things\n";
            std::this_thread::sleep_for(
                std::chrono::milliseconds(25));

            // check on kernel
            kernelCalculation(coro);
        }
        Data* results = coro->result;
        results->cleanup();
        printf("result[0]: %f\n", results->hostResult[0]);
        free(results->hostResult);
        delete coro;
    }
}
```

gather user input
launch kernels A, B

```
enum State {
    NotStarted = 0,
    Started = 1,
    AandBDone = 2,
    CDone = 3
};

// ... calculation(Coro* coro, State state){
// ... NotStarted:
// ... A = launchK...
// ... B = launchK...
// ... = State::Star...
// ... started:
// ... ntA->isDone...
// ... tC = launch...
// ... coro->eventB->getData(),
// ... = State::AandBDone;

case State::AandBDone:
    if(coro->eventC->isDone()){
        coro->result = coro->eventC->getData();
        coro->state = State::CDone;
    }
    return;
case State::CDone:
    return;
}
```

```
struct Coro {
    Coro(int x1, int x2): input1(x1), input2(x2),
        state(NotStarted){
        ~Coro(){
            delete eventA;
            delete eventB;
            delete eventC;
            delete result;
        }
        int input1;
        int input2;
        Event* eventA;
        Event* eventB;
        Event* eventC;
        Data *result;
        State state;
    };
};
```

Too much boilerplate!
Tedious to maintain!



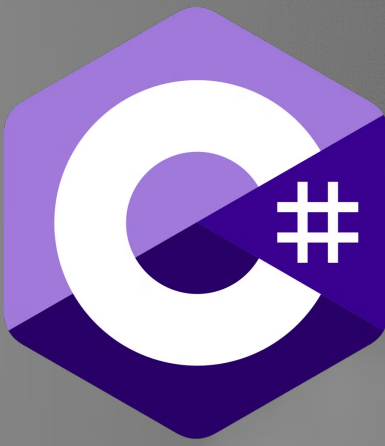
registers. Done!
launch c
other work

Async/Await Pattern

In 2012, C# introduced an asynchronous programming feature that delegated generation of the state machine and coroutine data structure to the compiler

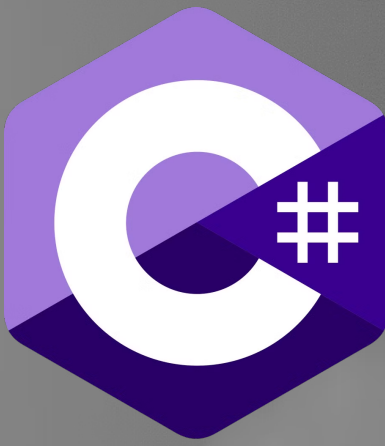
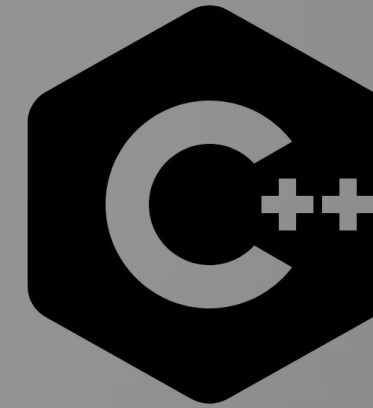
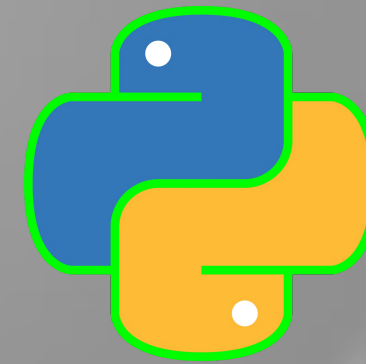
The feature was exposed through language keywords **async/await**.

async/await has become industry standard



Async/Await Pattern

- In 2012, C# introduced an asynchronous programming feature that delegated generation of the state machine and coroutine data structure to the compiler
- The feature was exposed through language keywords **async/await**.
- async/await has become industry standard
 - Python (2015)
 - Kotlin (2018) -> native or LLVM depending on if using jvm Kotlin or Kotlin/Native
 - Rust (2019) -> native (coroutines are lowered before llvm ir transformation)
 - C++ (2020) -> LLVM
 - Swift (2021) -> LLVM
 - Mojo (2024) -> LLVM or native?



02

LLVM Coroutine Lowering

LLVM Coroutine Lowerings

- **Switched-Resume Lowering**
 - Coroutine frame stored in handle
 - Asynchronous function $f: X \rightarrow R$ split into three functions: ramp ($f_ramp: X \rightarrow \text{Coro}$), resume ($r_resume: \text{Coro} \rightarrow ()$), destroy function ($f_destroy: \text{Coro} \rightarrow ()$)
 - Used by C++ 20
- **Returned-Continuation Lowering**
 - Coroutine frame is stored in a fixed size buffer specified by frontend
 - The suspendable function is lowered into a ramp function + resume functions for each suspension point. These resume functions return a function pointer to the subsequent resume function
 - Less commonly used
- **Async Lowering**
 - Coroutine frame is stored in the tail of a context argument
 - Async function is split into a ramp function and one resume function for each suspension point
 - Used by Swift



LLVM Coroutine Lowerings

- **Switched-Resume Lowering**

- Coroutine frame stored in handle
- Asynchronous function $f: X \rightarrow R$ split into three functions: ramp ($f_ramp: X \rightarrow \text{Coro}$), resume ($r_resume: \text{Coro} \rightarrow ()$), destroy function ($f_destroy: \text{Coro} \rightarrow ()$)
- Used by C++ 20

- **Returned-Continuation Lowering**

- Coroutine frame is stored in a fixed size buffer specified by frontend
- The suspendable function is lowered into a ramp function + resume functions for each suspension point. These resume functions return a function pointer to the subsequent resume function
- Less commonly used

- **Async Lowering**

- Coroutine frame is stored in the tail of a context argument
- Async function is split into a ramp function and one resume function for each suspension point
- Used by Swift




```
int main() {
    while(true){
        int x1, x2;
        std::cout << "Enter two integers: ";
        std::cin >> x1 >> x2;
        if(x1 < 0 || x2 < 0)
            break;
Task<std::pair<State,Data*>> task = kernelCalculation(x1,x2);
        State state = Started;
        while(!task.is_done()){
            // simulate other work
            if(task.result().first == AandBDone
                && state == Started)
                printf("finished A and B\n");
            state = task.result().first;
            std::cout << "tend to other things\n";
            std::this_thread::sleep_for(
                std::chrono::milliseconds(25));
task.resume();
        }
        Data* results = task.result().second;
        results->cleanup();
        printf("result[0]: %f\n", results->hostResult[0]);
        free(results->hostResult);
    }
}
```

```

int main() {
    while(true){
        int x1, x2;
        std::cout << "Enter two integers: ";
        std::cin >> x1 >> x2;
        if(x1 < 0 || x2 < 0)
            break;
        Task<std::pair<State,Data*>> task = kernelCalculation(x1,x2);
        State state = Started;
        while(!task.is_done()){
            // simulate other work
            if(task.result().first == AandBDone
                && state == Started)
                printf("finished A and B\n");
            state = task.result().first;
            std::cout << "tend to other things\n";
            std::this_thread::sleep_for(
                std::chrono::milliseconds(25));
            task.resume();
        }
        Data* results = task.result().second;
        results->cleanup();
        printf("result[0]: %f\n", results->hostResult[0]);
        free(results->hostResult);
    }
}

```

```

Task<std::pair<Data*,Data*>> kernelAandB(int x1, int x2){
    Event* eventA = launchKernelA(x1);
    Event* eventB = launchKernelB(x2);
    co_yield std::pair<Data*, Data*>(nullptr, nullptr);
    while(true){
        if(eventA->isDone() && eventB->isDone()){
            Data* a = eventA->getData();
            Data* b = eventB->getData();
            delete eventA;
            delete eventB;
            co_return std::pair<Data*, Data*>(a, b);
        }
        co_yield std::pair<Data*, Data*>(nullptr, nullptr);
    }
}

```

```

Task<std::pair<State,Data*>> kernelCalculation(int x1, int x2) {
    Task<std::pair<Data*,Data*>> result = kernelAandB(x1, x2);
    while(!result.is_done())
        co_await result;
    Event* eventC = launchKernelC(result.result().first,
        result.result().second);
    co_yield std::pair<State, Data*>(State::AandBDone, nullptr);
    while(true){
        if(eventC->isDone()){
            Data* result = eventC->getData();
            delete eventC;
            co_return std::pair<State, Data*>(State::CDone, result);
        }
        co_yield std::pair<State, Data*>(State::AandBDone, nullptr);
    }
}

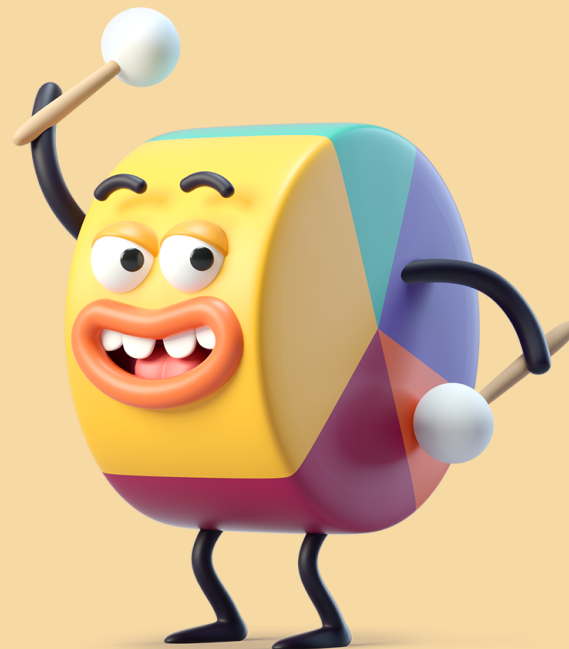
```



```

int main() {
    while(true){
        int x1, x2;
        std::cout << "Enter two integers: ";
        std::cin >> x1 >> x2;
        if(x1 < 0 || x2 < 0)
            break;
        Task<std::pair<State,Data*>> task = kernelCalculation(x1,x2);
        State state = Started;
        while(!task.is_done()){
            // simulate other work
            if(task.result().first == AandBDone
                && state == Started)
                printf("finished A and B\n");
            state = task.result().first;
            std::cout << "tend to other things\n";
            std::this_thread::sleep_for(
                std::chrono::milliseconds(25));
            task.resume();
        }
        Data* results = task.result().second;
        results->cleanup();
        printf("result[0]: %f\n", results->hostResult[0]);
        free(results->hostResult);
    }
}

```



```

Task<std::pair<Data*,Data*>> kernelAandB(int x1, int x2){
    Event* eventA = launchKernelA(x1);
    Event* eventB = launchKernelB(x2);
    co_yield std::pair<Data*, Data*>(nullptr, nullptr);
    while(true){
        if(eventA->isDone() && eventB->isDone()){
            Data* a = eventA->getData();
            Data* b = eventB->getData();
            delete eventA;
            delete eventB;
            co_return std::pair<Data*, Data*>(a, b);
        }
        co_yield std::pair<Data*, Data*>(nullptr, nullptr);
    }
}

Task<std::pair<State,Data*>> kernelCalculation(int x1, int x2) {
    Task<std::pair<Data*,Data*>> result = kernelAandB(x1, x2);
    while(!result.is_done())
        co_await result;
    Event* eventC = launchKernelC(result.result().first,
        result.result().second);
    co_yield std::pair<State, Data*>(State::AandBDone, nullptr);
    while(true){
        if(eventC->isDone()){
            Data* result = eventC->getData();
            delete eventC;
            co_return std::pair<State, Data*>(State::CDone, result);
        }
        co_yield std::pair<State, Data*>(State::AandBDone, nullptr);
    }
}

```

Simple Example

```
Task<int> coroutine_example(int x1, int x2) {  
    int intermediate = x1 * x2;  
    printf("%d\n", intermediate);  
    int final = intermediate * intermediate;  
    co_await std::suspend_always();  
    co_return final;  
}
```

```
int main() {  
    int x1, x2;  
    std::cout << "Enter two integers: ";  
    std::cin >> x1 >> x2;  
    Task<int> task = coroutine_example(x1, x2);  
    task.resume();  
    printf("%d\n", task.result());  
    return 0;  
}
```

```
template<typename T>  
class Task {  
public:  
    struct promise_type {  
        T value;  
        // methods here  
    };  
  
    using handle_type =  
        std::coroutine_handle<promise_type>;  
  
    Task(handle_type h) : handle(h) {}  
    ~Task() {  
        if (handle) handle.destroy();  
    }  
    // methods here  
private:  
    handle_type handle;  
};
```


Simple Example

```
Task<int> coroutine_example(int x1, int x2) {
    int intermediate = x1 * x2;
    printf("%d\n", intermediate);
    int final = intermediate * intermediate;
    co_await std::suspend_always();
    co_return final;
}

int main() {
    int x1, x2;
    std::cout << "Enter two integers: ";
    std::cin >> x1 >> x2;
    Task<int> task = coroutine_example(x1, x2);
    task.resume();
    printf("%d\n", task.result());
    return 0;
}
```

```
template<typename T>
class Task {
public:
    struct promise_type {
        T value;
        // methods here
    };

    using handle_type =
        std::coroutine_handle<promise_type>;

    Task(handle_type h) : handle(h) {}
    ~Task() {
        if (handle) handle.destroy();
    }
    // methods here
private:
    handle_type handle;
};
```

Simple Example

```
Task<int> coroutine_example(int x1, int x2) {  
    int intermediate = x1 * x2;  
    printf("%d\n", intermediate);  
    int final = intermediate * intermediate;  
    co_await std::suspend_always();  
    co_return final;  
}
```

```
int main() {  
    int x1, x2;  
    std::cout << "Enter two integers: ";  
    std::cin >> x1 >> x2;  
    Task<int> task = coroutine_example(x1, x2);  
    task.resume();  
    printf("%d\n", task.result());  
    return 0;  
}
```

```
template<typename T>  
class Task {  
public:  
    struct promise_type {  
        T value;  
        // methods here  
    };  
  
    using handle_type =  
        std::coroutine_handle<promise_type>;  
  
    Task(handle_type h) : handle(h) {}  
    ~Task() {  
        if (handle) handle.destroy();  
    }  
    // methods here  
private:  
    handle_type handle;  
};
```


Simple Example

```
Task<int> coroutine_example(int x1, int x2) {  
    int intermediate = x1 * x2;  
    printf("%d\n", intermediate);  
    int final = intermediate * intermediate;  
    co_await std::suspend_always();  
    co_return final;  
}
```

```
int main() {  
    int x1, x2;  
    std::cout << "Enter two integers: ";  
    std::cin >> x1 >> x2;  
    Task<int> task = coroutine_example(x1, x2);  
    task.resume();  
    printf("%d\n", task.result());  
    return 0;  
}
```

```
template<typename T>  
class Task {  
public:  
    struct promise_type {  
        T value;  
        // methods here  
    };  
};
```

```
using handle_type =  
std::coroutine_handle<promise_type>;
```

```
Task(handle_type h) : handle(h) {}  
~Task() {  
    if (handle) handle.destroy();  
}  
    // methods here  
private:  
    handle_type handle;  
};
```

Simple Example: Coroutine Frame

-O3 -fcoroutines -std=c++20

```
Task<int> coroutine_example(int x1, int x2) {  
    int intermediate = x1 * x2;  
    printf("%d\n", intermediate);  
    int final = intermediate * intermediate;  
    co_await std::suspend_always();  
    co_return final;  
}
```

```
%class.Task = type { %"struct.std::_n4861::coroutine_handle" }  
%"struct.std::_n4861::coroutine_handle" = type { ptr }  
%coroutine_example(int, int).Frame = type { ptr, ptr,  
%"struct.Task<int>::promise_type", i32, i32, i1 }  
%"struct.Task<int>::promise_type" = type { i32 }
```


Simple Example: Coroutine Frame

```
Task<int> coroutine_example(int x1, int x2) {  
    int intermediate = x1 * x2;  
    printf("%d\n", intermediate);  
    int final = intermediate * intermediate;  
    co_await std::suspend_always();  
    co_return final;  
}
```

```
%class.Task = type { %"struct.std::_n4861::coroutine_handle" }  
%"struct.std::_n4861::coroutine_handle" = type { ptr }  
%coroutine_example(int, int).Frame = type { ptr, ptr,  
%"struct.Task<int>::promise_type", i32, i32, i1 }  
%"struct.Task<int>::promise_type" = type { i32 }
```

Simple Example: Coroutine Frame

-O3 -fcoroutines -std=c++20

```
Task<int> coroutine_example(int x1, int x2) {  
    int intermediate = x1 * x2;  
    printf("%d\n", intermediate);  
    int final = intermediate * intermediate;  
    co_await std::suspend_always();  
    co_return final;  
}
```

```
%class.Task = type { %"struct.std::_n4861::coroutine_handle" }  
%"struct.std::_n4861::coroutine_handle" = type { ptr }  
%coroutine_example(int, int).Frame = type { ptr, ptr,  
%"struct.Task<int>::promise_type" i32, i32, i1 }  
%"struct.Task<int>::promise_type" = type { i32 }
```


Simple Example: Coroutine Frame

```
Task<int> coroutine_example(int x1, int x2) {  
    int intermediate = x1 * x2;  
    printf("%d\n", intermediate);  
    int final = intermediate * intermediate;  
    co_await std::suspend_always();  
    co_return final;  
}
```

```
%class.Task = type { %"struct.std::_n4861::coroutine_handle" }  
%"struct.std::_n4861::coroutine_handle" = type { ptr }  
%coroutine_example(int, int).Frame = type { ptr, ptr,  
%"struct.Task<int>::promise_type", i32, i32, i1 }  
%"struct.Task<int>::promise_type" = type { i32 }
```

Simple Example: Coroutine Frame

```
Task<int> coroutine_example(int x1, int x2) {  
    int intermediate = x1 * x2;  
    printf("%d\n", intermediate);  
    int final = intermediate * intermediate;  
    co_await std::suspend_always();  
    co_return final;  
}
```

```
%class.Task = type { %"struct.std::_n4861::coroutine_handle" }  
%"struct.std::_n4861::coroutine_handle" = type { ptr }  
%coroutine_example(int, int).Frame = type { ptr, ptr,  
%"struct.Task<int>::promise_type", i32, i32, i1 }  
%"struct.Task<int>::promise_type" = type { i32 }
```

Simple Example: Ramp/Resume

-O3 -fcoroutines -std=c++20

```
Task<int> coroutine_example(int x1, int x2) {  
    int intermediate = x1 * x2;  
    printf("%d\n", intermediate);  
    int final = intermediate * intermediate;  
    co_await std::suspend_always();  
    co_return final;  
}
```

```
define dso_local void @coroutine_example(int, int)(ptr dead_on_unwind noreturn writable writeonly sret(%class.Task) align 8  
%agg.result, i32 noundef %x1, i32 noundef %x2) local_unnamed_addr personality ptr @_gxx_personality_v0 {  
entry:  
    %call = tail call noalias noundef nonnull dereferenceable(32) ptr @operator_new(unsigned long)(i64 noundef 32) #11  
    store ptr @coroutine_example(int, int) (.resume), ptr %call, align 8  
    %destroy.addr = getelementptr inbounds %coroutine_example(int, int).Frame, ptr %call, i64 0, i32 1  
    store ptr @coroutine_example(int, int) (.destroy), ptr %destroy.addr, align 8  
    %x2.spill.addr = getelementptr inbounds %coroutine_example(int, int).Frame, ptr %call, i64 0, i32 4  
    store i32 %x2, ptr %x2.spill.addr, align 4  
    %x1.spill.addr = getelementptr inbounds %coroutine_example(int, int).Frame, ptr %call, i64 0, i32 3  
    store i32 %x1, ptr %x1.spill.addr, align 4  
    store ptr %call, ptr %agg.result, align 8  
    %mul = mul nsw i32 %x2, %x1  
    %call13 = tail call i32 @printf(ptr noundef nonnull dereferenceable(1) @.str, i32 noundef %mul)  
    %index.addr78 = getelementptr inbounds %coroutine_example(int, int).Frame, ptr %call, i64 0, i32 5  
    store i1 false, ptr %index.addr78, align 1  
    ret void  
}  
define internal fastcc void @coroutine_example(int, int) (.resume)(ptr noreturn noundef nonnull align 8  
dereferenceable(32) %0) personality ptr @_gxx_personality_v0 {  
CoroEnd:  
    %_promise.reload.addr = getelementptr inbounds %coroutine_example(int, int).Frame, ptr %0, i64 0, i32 2  
    %x2.reload.addr = getelementptr inbounds %coroutine_example(int, int).Frame, ptr %0, i64 0, i32 4  
    %x2.reload = load i32, ptr %x2.reload.addr, align 8  
    %x1.reload.addr = getelementptr inbounds %coroutine_example(int, int).Frame, ptr %0, i64 0, i32 3  
    %x1.reload = load i32, ptr %x1.reload.addr, align 4  
    %mul77 = mul nsw i32 %x1.reload, %x2.reload  
    %mul14 = mul nsw i32 %mul77, %mul77  
    store i32 %mul14, ptr %_promise.reload.addr, align 8  
    store ptr null, ptr %0, align 8  
    ret void  
}
```


Simple Example: Coroutine Frame O0

-O0 -fcoroutines -std=c++20

```
Task<int> coroutine_example(int x1, int x2) {  
    int intermediate = x1 * x2;  
    printf("%d\n", intermediate);  
    int final = intermediate * intermediate;  
    co_await std::suspend_always();  
    co_return final;  
}
```

```
%coroutine_example(int, int).Frame = type { ptr, ptr, %"struct.Task<int>::promise_type" i32, i32, i32, i32, i2,  
%"struct.std::_n4861::suspend_never", %"struct.std::_n4861::suspend_always", %"struct.std::_n4861::suspend_always" }
```

Simple Example: Coroutine Frame 00

-O0 -fcoroutines -std=c++20

```
Task<int> coroutine_example(int x1, int x2) {  
    int intermediate = x1 * x2;  
    printf("%d\n", intermediate);  
    int final = intermediate * intermediate;  
    co_await std::suspend_always();  
    co_return final;  
}
```

```
%coroutine_example(int, int).Frame = type { ptr, ptr, %"struct.Task<int>::promise_type", i32, i32, i32, i32, i2,  
%"struct.std::_n4861::suspend_never", %"struct.std::_n4861::suspend_always", %"struct.std::_n4861::suspend_always" }
```

Simple Example: Frame Allocation

-O3 -fcoroutines -std=c++20

```
Task<int> coroutine_example(int x1, int x2) {
    int intermediate = x1 * x2;
    printf("%d\n", intermediate);
    int final = intermediate * intermediate;
    co_await std::suspend_always();
    co_return final;
}
```

```
int main() {
    int x1, x2;
    std::cout << "Enter two integers: ";
    std::cin >> x1 >> x2;
    Task<int> task = coroutine_example(x1, x2);
    task.resume();
    printf("%d\n", task.result());
    return 0;
}
```

```
define dso_local void @coroutine_example(int, int)(ptr dead_on_unwind nocapture writable writeonly sret(%class.Task) align 8
%agg.result, i32 noundef %x1, i32 noundef %x2) local_unnamed_addr personality ptr @_gxx_personality_v0 {
entry:
```

```
    %call = tail call noalias noundef nonnull dereferenceable(32) ptr @operator_new(unsigned long)(i64 noundef 32) #11
```

```
    store ptr @coroutine_example(int, int) (.resume), ptr %call, align 8
    %destroy.addr = getelementptr inbounds %coroutine_example(int, int).Frame, ptr %call, i64 0, i32 1
    store ptr @coroutine_example(int, int) (.destroy), ptr %destroy.addr, align 8
    %x2.spill.addr = getelementptr inbounds %coroutine_example(int, int).Frame, ptr %call, i64 0, i32 4
    store i32 %x2, ptr %x2.spill.addr, align 4
    %x1.spill.addr = getelementptr inbounds %coroutine_example(int, int).Frame, ptr %call, i64 0, i32 3
    store i32 %x1, ptr %x1.spill.addr, align 4
    store ptr %call, ptr %agg.result, align 8
    %mul = mul nsw i32 %x2, %x1
    %call13 = tail call i32 (ptr, ...) @printf(ptr noundef nonnull dereferenceable(1) @.str, i32 noundef %mul)
    %index.addr78 = getelementptr inbounds %coroutine_example(int, int).Frame, ptr %call, i64 0, i32 5
    store i1 false, ptr %index.addr78, align 1
    ret void
}
```

```
define internal fastcc void @coroutine_example(int, int) (.destroy)(ptr noundef nonnull align 8 dereferenceable(32) %0)
personality ptr @_gxx_personality_v0 {
resume entry:
```

```
    tail call void @operator_delete(void*)(ptr noundef nonnull %0) #12
```

```
    ret void
}
```


Is Switch-Resume Performant enough?



- **What is the memory footprint?**



Is Switch-Resume Performant enough?



- **What is the memory footprint?** All values are stored in the frame, leading to bloated frame sizes and unnecessary store ops ❌



Is Switch-Resume Performant enough?



- **What is the memory footprint?** All values are stored in the frame, leading to bloated frame sizes and unnecessary store ops ✗
- **How are coroutines allocated?**



Is Switch-Resume Performant enough?

M

- **What is the memory footprint?** All values are stored in the frame, leading to bloated frame sizes and unnecessary store ops ✗
- **How are coroutines allocated?** It appears that LLVM failed to promote our heap allocation ✗



Is Switch-Resume Performant enough?



- **What is the memory footprint?** All values are stored in the frame, leading to bloated frame sizes and unnecessary store ops ✗
- **How are coroutines allocated?** It appears that LLVM failed to promote our heap allocation ✗
- **What is the IR footprint?**



Is Switch-Resume Performant enough?



- **What is the memory footprint?** All values are stored in the frame, leading to bloated frame sizes and unnecessary store ops ✗
- **How are coroutines allocated?** It appears that LLVM failed to promote our heap allocation ✗
- **What is the IR footprint?** In O0, four functions are generated (ramp, resume, destroy, cleanup). We don't need cleanup or destroy functions because Mojo has asap destruction that will mark the locations for destruction of frame values and the coroutine itself ✗



Is Switch-Resume Performant enough?

M

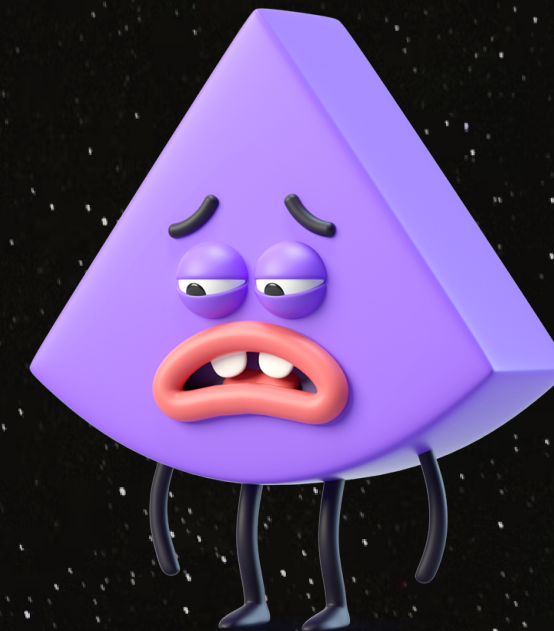
- **What is the memory footprint?** All values are stored in the frame, leading to bloated frame sizes and unnecessary store ops ✖
- **How are coroutines allocated?** It appears that LLVM failed to promote our heap allocation ✖
- **What is the IR footprint?** In O0, four functions are generated (ramp, resume, destroy, cleanup). We don't need cleanup or destroy functions because Mojo has asap destruction that will mark the locations for destruction of frame values and the coroutine itself ✖
- **What is the default behavior of LLVM coroutines?** In O3 the first state was moved into the ramp function but despite this we observed a bloated frame size and duplicate operations. ⚠



Is Switch-Resume Performant enough?

M

- **What is the memory footprint?** All values are stored in the frame, leading to bloated frame sizes and unnecessary store ops ✗
- **How are coroutines allocated?** It appears that LLVM failed to promote our heap allocation ✗
- **What is the IR footprint?** In O0, four functions are generated (ramp, resume, destroy, cleanup). We don't need cleanup or destroy functions because Mojo has asap destruction that will mark the locations for destruction of frame values and the coroutine itself ✗
- **What is the default behavior of LLVM coroutines?** In O3 the first state was moved into the ramp function but despite this we observed a bloated frame size and duplicate operations. 🚧



LLVM Coroutine Lowerings

- **Switched-Resume Lowering**

- Coroutine frame stored in handle
- Asynchronous function $f: X \rightarrow R$ split into three functions: ramp ($f_ramp: X \rightarrow \text{Coro}$), resume ($r_resume: \text{Coro} \rightarrow ()$), destroy function ($f_destroy: \text{Coro} \rightarrow ()$)
- Used by C++ 20

- **Returned-Continuation Lowering**

- Coroutine frame is stored in a fixed size buffer specified by frontend
- The suspendable function is lowered into a ramp function + resume functions for each suspension point. These resume functions return a function pointer to the subsequent resume function
- Less commonly used

- **Async Lowering**

- Coroutine frame is stored in the tail of a context argument
- Async function is split into a ramp function and one resume function for each suspension point
- Used by Swift



Async Lowering: Coroutine Frame

```
swiftc -parse-as-library -emit-ir foo.swift -o foo.ll
```

```
func asyncFunction(x1:Int, x2:Int) async -> Int {  
  let intermediate = x1 * x2  
  print(intermediate)  
  let final = intermediate * intermediate  
  await Task.yield()  
  return final  
}
```

```
@main  
struct MainProgram {  
  static func main() async {  
    let x1 = Int(readLine() ?? "0") ?? 0  
    let x2 = Int(readLine() ?? "1") ?? 1  
    let finalResult = await asyncFunction(x1: x1, x2: x2)  
    print("Result: \(finalResult)")  
  }  
}
```

```
%"$s3foo13asyncFunction2x12x2S2i_SitYaF.Frame" = type {  
  ptr, i64, i64, i64, i64, i64, i64, i64, ptr }
```

Async Lowering: Coroutine Frame

```
swiftc -parse-as-library -emit-ir -foo.swift -o foo.ll
```

```
func asyncFunction(x1:Int, x2:Int) async -> Int {  
    let intermediate = x1 * x2  
    print(intermediate)  
    let notUsedAcrossSuspoints = intermediate * x1  
    print(notUsedAcrossSuspoints)  
    let final = intermediate * intermediate  
    await Task.yield()  
    return final  
}  
  
@main  
struct MainProgram {  
    static func main() async {  
        let x1 = Int(readLine() ?? "0") ?? 0  
        let x2 = Int(readLine() ?? "1") ?? 1  
        let finalResult = await asyncFunction(x1: x1, x2: x2)  
        print("Result: \(finalResult)")  
    }  
}
```

```
%"$s3foo13asyncFunction2x12x2S2i_SitYaF.Frame" = type {  
    ptr, i64, i64, i64, i64, i64, i64, i64, i64, ptr }
```

// excerpt from resume function

```
%notUsedAcrossSuspoints = getelementptr inbounds  
%"$s4foo213asyncFunction2x12x2S2i_SitYaF.Frame", ptr  
%async.ctx.frameptr, i32 0, i32 4
```

```
:  
store i64 %9, ptr %notUsedAcrossSuspoints, align 8
```

Async Lowering: Resume Functions

```
swiftc -O -parse-as-library -emit-ir foo.swift -o foo.ll
```

```
func asyncFunction(x1:Int, x2:Int) async -> Int {
  let intermediate = x1 * x2
  print(intermediate)
  let notUsedAcrossSuspoints = intermediate * x1
  print(notUsedAcrossSuspoints)
  let final = intermediate * intermediate
  await Task.yield()
  return final
}

@main
struct MainProgram {
  static func main() async {
    let x1 = Int(readLine() ?? "0") ?? 0
    let x2 = Int(readLine() ?? "1") ?? 1
    let finalResult = await asyncFunction(x1: x1, x2: x2)
    print("Result: \(finalResult)")
  }
}
```

```
define hidden swifttailcc void
@$s3foo13asyncFunction2x12x2S2i_SitYaF"(ptr swiftasync
%0, i64 %1, i64 %2) #0 {
:
}

define internal swifttailcc void
@$s3foo13asyncFunction2x12x2S2i_SitYaFTY0_"(ptr
swiftasync %0) #1
:

define internal swifttailcc void
@$s3foo13asyncFunction2x12x2S2i_SitYaFTQ1_"(ptr
nocapture readonly swiftasync %0)
:
```


A Historical Note 🦴

- Mojo async started off using Async Lowering! But....
 - we observed inactive values being stored on the frame
 - we observed allocations not getting optimized away
 - the coro pass depends on multiple functions living in the same module, preventing parallelization efforts

M



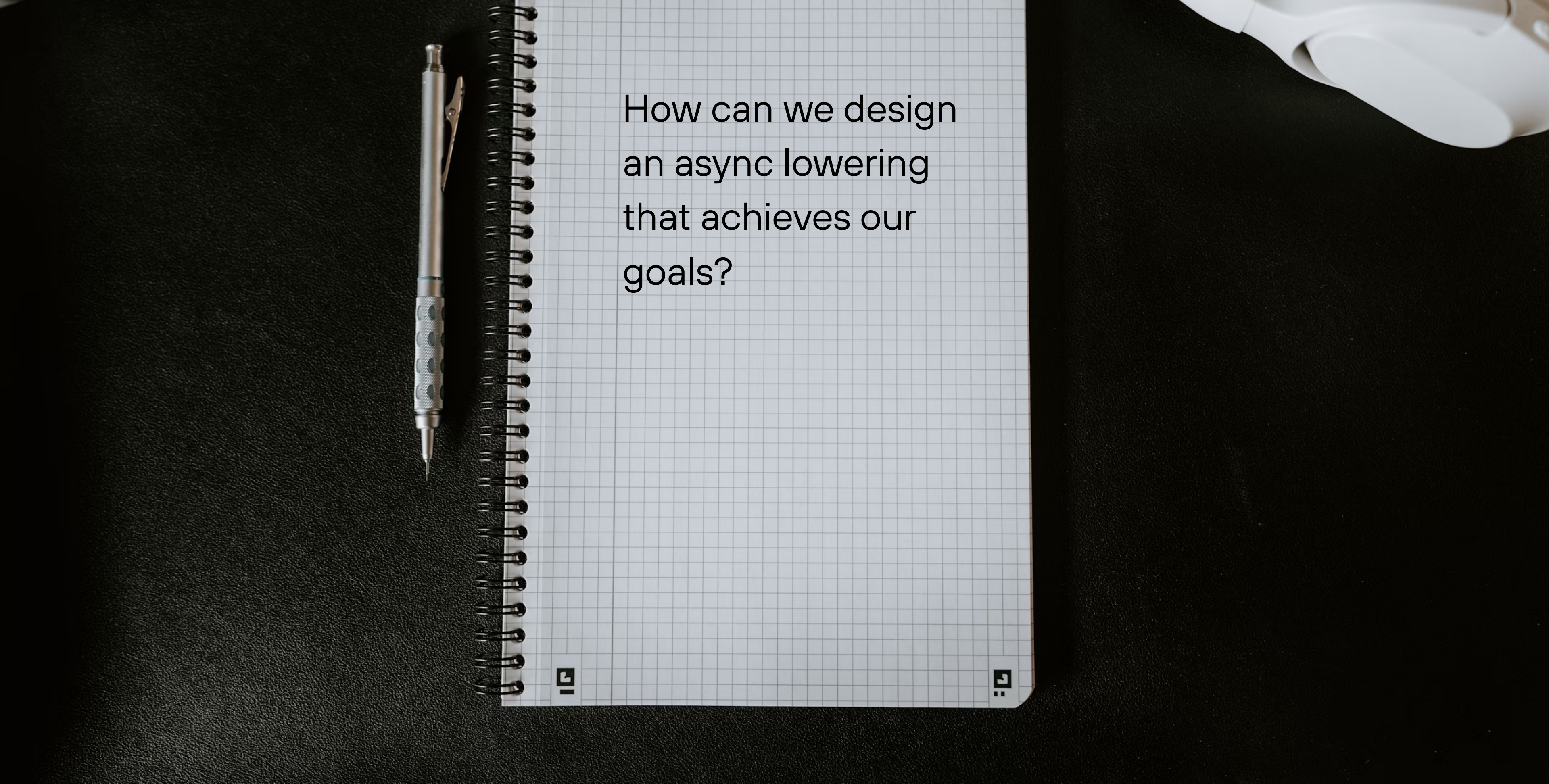
A Historical Note 🦴

We want

- Finer grained control over the size of the frame
- Control over the number of functions generated
- Control of the order of lowerings so we can optimize allocations

M



A photograph of a workspace. In the center is a spiral-bound notebook with a white grid-patterned cover. A silver ballpoint pen with a textured grip lies vertically to the left of the notebook. To the right of the notebook, the corner of a white computer mouse is visible. The entire scene is set against a solid black background.

How can we design
an async lowering
that achieves our
goals?

A picture may be worth 1,000 words, but a caption is the cherry on top.

03

MLIR Coroutine Implementation

Async Support in MLIR

M

- 01 **Define the Dialect**
- 02 Identify Order of Lowerings
- 03 Define the transformations



Async Support in MLIR

- 01 Define the Dialect
- 02 **Identify Order of Lowerings**
- 03 Define the transformations

M



Async Support in MLIR

- 01 Define the Dialect
- 02 Identify Order of Lowerings
- 03 **Define the transformations**

M



Define the Dialect

Ops to represent the yet-to-be generated functions	Ops to represent access to members of the yet to be generated frame
• co.invoke to model calls to an async function. • co.suspend to model a suspension point • co.await to model spawning a child coroutine from a parent coroutine. Implicit suspension point, so why not just use the suspend op? It's embedded in the mojo async library code and alleviates the need to set the parent coroutine, which would require us to create a representation of the coroutine handle in the IR. • co.destroy to model the destruction of a coroutine. Generally not intended to be embedded in mojo code but generated by the passes that handle value lifetimes	• co.resume, co.get_callback_ptr need to be exposed to the language so that we can write runtime libraries. For example, we may want to assign a worker thread to the resume function of a coroutine and assign a closure to invoke upon completion of the coroutine. These ops provide a means to access these components before the compiler generates them. • co.set_byref_error_result allows the caller to allocate memory for the result and error and set the address of this location in the coroutine. This enables coroutines to return non-movable types • co.get_results - generally coroutines only return results by reference but there are cases where asynchronous functions return a value. For example, raising functions return a boolean value that indicates whether or not the function exited without error. In this case the value is stored in a separate field right before the frame in a coroutine. Access to this value is exposed via this op

Order of Lowerings

- Support both hot and cold started coroutines (optimization)
- Promote heap allocated coroutines to stack (optimization)
- Simplify state machine lowering (requirement on control flow representation)



Support Both Hot- and Cold-Started Coroutines

- **Hot started coroutine** is a coroutine whose ramp function includes the first state of the coroutine.
- **Cold started coroutine** is a coroutine whose ramp function only instantiates the coroutine and returns it; no code from the original asynchronous function definition is executed.

Eligible to be hot started:	Not eligible to be hot started:
<pre>@no_inline async fn print_it(x: Int) -> Int: print(x) return 1 @no_inline async fn hot_invoke(init: Int): var a = init a += await print_it(a) print(a)</pre>	<pre>@no_inline async fn print_it(x: Int) -> Int: print(x) return 1 @no_inline async fn cold_invoke(init: Int): var a = init var task = create_task(print_it(a)) a += await task print(a)</pre>

Support Both Hot- and Cold-Started Coroutines

- **Hot started coroutine** is a coroutine whose ramp function includes the first state of the coroutine.
- **Cold started coroutine** is a coroutine whose ramp function only instantiates the coroutine and returns it; no code from the original asynchronous function definition is executed.

Eligible to be hot started:	Not eligible to be hot started:
<pre>@no_inline async fn print_it(x: Int) -> Int: print(x) return 1</pre> <pre>@no_inline async fn hot_invoke(init: Int): var a = init a += await print_it(a) print(a)</pre>	<pre>@no_inline async fn print_it(x: Int) -> Int: print(x) return 1</pre> <pre>@no_inline async fn cold_invoke(init: Int): var a = init var task = create_task(print_it(a)) a += await task print(a)</pre>

how do we
generate hot
versus cold
coroutines?

Support Both Hot- and Cold-Started Coroutines

- **Hot started coroutine** is a coroutine whose ramp function includes the first state of the coroutine.
- **Cold started coroutine** is a coroutine whose ramp function only instantiates the coroutine and returns it; no code from the original asynchronous function definition is executed.

Hot start eligible IR	After canonicalization is applied
<pre>%3 = co.invoke[(index, !kgen.pointer<index> byref_result) async -> !kgen.none: @"print_it"](%2) %4 = co.await %3, %7 : (!co.routine, !kgen.pointer<index>) -> !kgen.none</pre>	<pre>%4 = co.hot_invoke[(index, !kgen.pointer<index> byref_result) async -> !kgen.none: @"async::print_it(::Int)"](%2, %3)</pre>

Support Both Hot- and Cold-Started Coroutines

Lowering Plan
Parser: Generate co.await/co.invoke
Canonicalization: replace co.await(co.invoke) with co.hot_invoke
– TODO: when do we lower the co dialect ops? –

Elide Heap Allocated Coroutines When Possible

This is an optimization but it's an important consideration during the design phase because we have to identify where our lowerings go.

We can only elide the heap allocation if the coroutine does not escape. The coroutine always escapes the ramp function because that is where it is created. **If we inline the ramp function before heap to stack pass then the coroutine becomes eligible for ellison.** The updated plan is then:

Lowering Plan
Parser: Generate co.await/co.invoke
Canonicalization: replace co.await(co.invoke) with co.hot_invoke
Lower async functions into ramp + resume (+ lower all co dialect ops)
Inline ramp function (run existing inliner)
Perform heap to stack promotion

Simplify State Machine Lowering

```
async fn cf(init: Int):  
  var a = 0  
  while a < 10:  
    print(a)  
    if a % 2 == 0:  
      a += await print_it(a)  
    print(a)  
    a = a - 1
```

```
kgen.generator @cf(%arg0: !Int, %arg1: !lit.ref<none, mut #lit.any.origin> byref_result) async -> {  
  // declare 'a'  
  hlcf.loop "_loop_0" {  
    %6 = /* result of comparing a to 10 */  
    hlcf.if %6 {  
      hlcf.yield  
    } else {  
      hlcf.break "_loop_0"  
    }  
    %12 = /* result of comparing a % 2 to 0 */  
    hlcf.if %12 {  
      %13 = lit.ref.load %1 : <!!Int, mut #lit.any.origin>  
      %14 = co.invoke[(!Int, !lit.ref<!!Int, mut #lit.any.origin> byref_result) async -> !kgen.none: @"foo::print_it(::Int)"](%13)  
      // await %14  
      %24 = kgen.call @ foo::print(::Int) (%23) : (!Int) -> !kgen.none  
      hlcf.yield  
    } else {  
      hlcf.yield  
    }  
    hlcf.continue  
  }  
  kgen.return  
}
```

Simplify State Machine Lowering

```
async fn cf(init: Int):  
  var a = 0  
  while a < 10:  
    print(a)  
    if a % 2 == 0:  
      a += await print_it(a)  
    print(a)  
    a = a - 1
```

```
llvm.func internal @"async::cf(::Int)_resume"(%arg0: !llvm.ptr {!llvm.noundef}) {  
  :  
  llvm.br ^bb1  
^bb1: // 2 preds: ^bb0, ^bb7  
  :  
  llvm.cond_br %12, ^bb2, ^bb3  
^bb2: // pred: ^bb1  
  :  
  llvm.br ^bb4  
^bb3: // pred: ^bb1  
  :  
  llvm.br ^bb8  
^bb4: // pred: ^bb2  
  :  
  llvm.cond_br %24, ^bb5, ^bb6  
^bb5: // pred: ^bb4  
  :  
  co.suspend {  
    :  
    %93 = llvm.call @"async::print_it(::Int)_hot_ramp"(%92, %arg0, %26, %90) : (!llvm.ptr, !llvm.ptr, i64, !llvm.ptr) -> !llvm.ptr  
    :  
    co.suspend.end  
  }  
  :  
  llvm.br ^bb7  
^bb6: // pred: ^bb4  
  llvm.br ^bb7  
^bb7: // 2 preds: ^bb5, ^bb6  
  llvm.br ^bb1  
^bb8: // pred: ^bb3  
  :
```


Simplify State Machine Lowering

Lowering Plan
Parser: Generate co.await/co.invoke
Canonicalization: replace co.await(co.invoke) with co.hot_invoke
Lower async functions into ramp + resume (+ lower all co dialect ops)
Inline ramp function (run existing inliner)
Perform heap to stack promotion
Lower Control Flow
Lower Suspension Points (i.e. state machine transformation)

Asynchronous Function Lowering

Lowering Plan
Parser: Generate co.await/co.invoke
Canonicalization: replace co.await(co.invoke) with co.hot_invoke
Lower async functions into ramp + resume (+ lower all co dialect ops)
Inline ramp function (run existing inliner)
Perform heap to stack promotion
Lower Control Flow
Lower Suspension Points (i.e. state machine transformation)

Overview of Async Function Lowering

M

- For each async function, determine if it needs a hot ramp, a cold ramp, or both
- Preprocess the async function:
 - Move stack allocations as close to their first use as possible
 - Lower implicit suspension points
- Create coroutine for each asynchronous function
 - Calculate the frame (hot cold or both)
 - Generate the ramp function
 - Generate the resume function
- Replace hot invokes/cold invokes with calls to corresponding ramp functions
- Lower all other coroutine dialect ops. Now that we have the frame types, these are more or less GetElementPtr ops



Overview of Async Function Lowering

M

- For each async function, determine if it needs a hot ramp, a cold ramp, or both
- Preprocess the async function:
 - Move stack allocations as close to their first use as possible
 - Lower implicit suspension points
- Create coroutine for each asynchronous function
 - Calculate the frame (hot cold or both)
 - Generate the ramp function
 - Generate the resume function
- Replace hot invokes/cold invokes with calls to corresponding ramp functions
- Lower all other coroutine dialect ops. Now that we have the frame types, these are more or less GetElementPtr ops



Prerequisite to Frame Calculation: Lower all implicit suspension points

- Frame calculation requires all suspension points to be modeled explicitly so it can identify which variables must be store into the frame
- Await represents a suspension point
- co.hot_invoke must be lowered before frame calculation also because it models an implicit suspension point since it represents a call to a ramp with the first state, which means it executes the coroutine up to its first suspension point

IR with await that was not lowered into hot invoke:	After Lower awaits:
<pre>kgen.func @foo(%arg0: !co.routine) -> i1 { %0 = co.await %arg0 : (!co.routine) -> i1 kgen.return %0 : i1 }</pre>	<pre>kgen.func @foo(%arg0: !co.routine) -> i1 { co.suspend (%hdl) { %1 = co.get_callback_ptr %arg0 : <struct<(!co.routine) -> (), !co.routine)>> %2 = kgen.struct.gep %1[0] : <struct<(!co.routine) -> (), !co.routine)>> %3 = kgen.struct.gep %1[1] : <struct<(!co.routine) -> (), !co.routine)>> %4 = co.resume %hdl : <(!co.routine) -> ()> pop.store %4, %2 : !kgen.pointer<(!co.routine) -> ()> pop.store %hdl, %3 : !kgen.pointer<!co.routine> %5 = co.resume %arg0 : <(!co.routine) -> ()> kgen.call_indirect %5(%arg0) : (!co.routine) -> () co.suspend.end } %0 = co.get_results %arg0 : i1 co.destroy %arg0 kgen.return %0 : i1 }</pre>

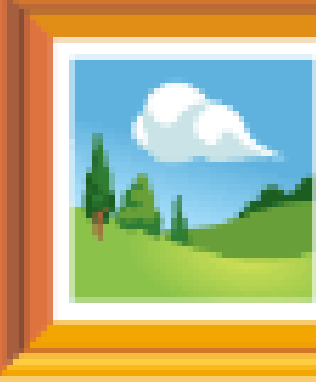
Overview of Async Function Lowering

M

- For each async function, determine if it needs a hot ramp, a cold ramp, or both
- Preprocess the async function:
 - Move stack allocations as close to their first use as possible
 - Lower implicit suspension points
- Create coroutine for each asynchronous function
 - Calculate the frame (hot cold or both)
 - Generate the ramp function
 - Generate the resume function
- Replace hot invokes/cold invokes with calls to corresponding ramp functions
- Lower all other coroutine dialect ops. Now that we have the frame types, these are more or less GetElementPtr ops

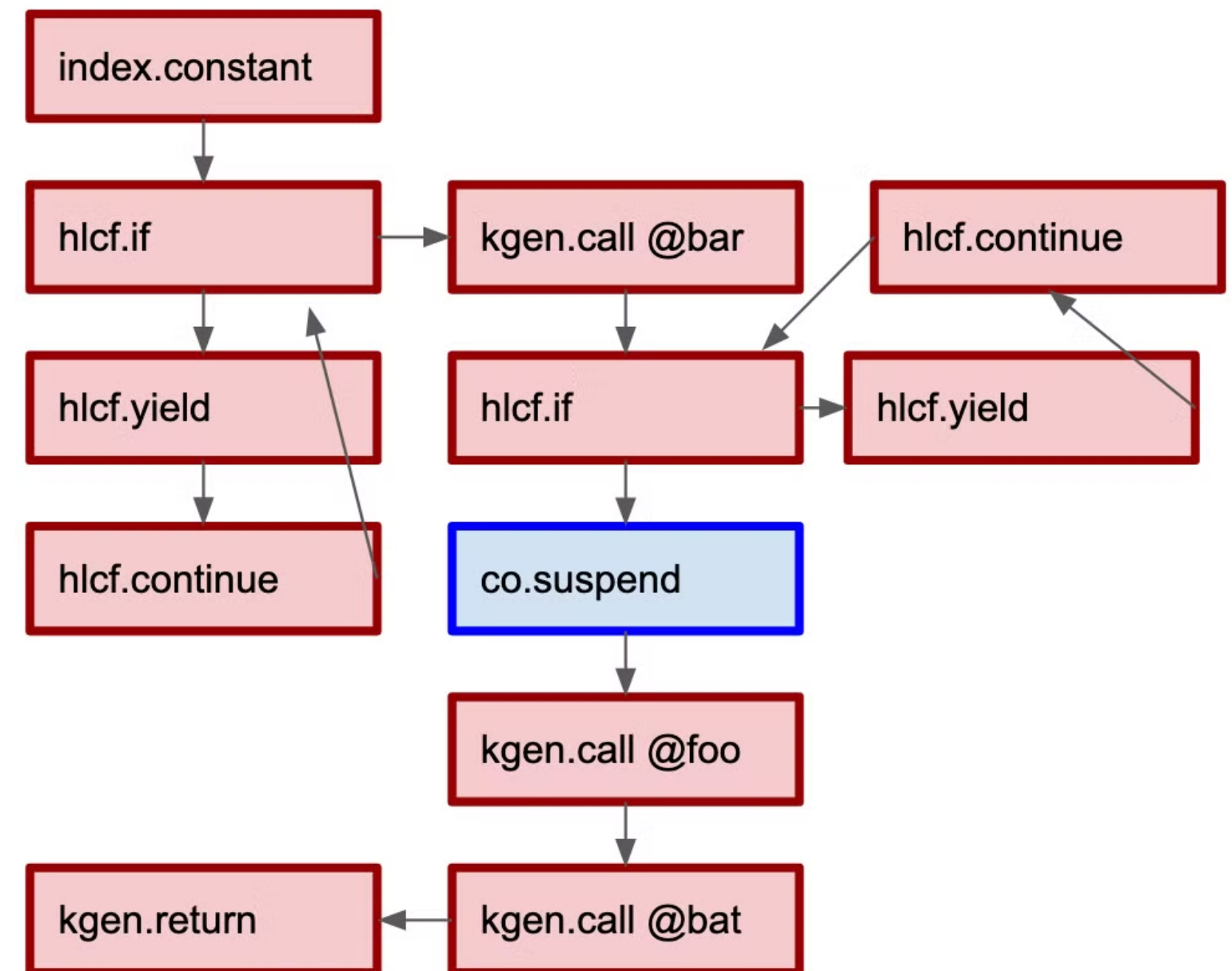


Frame Challenges

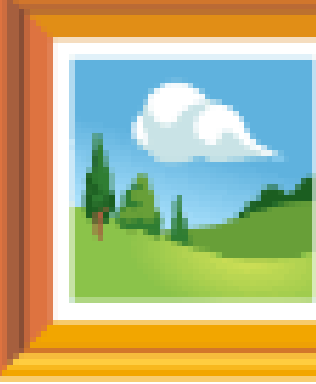


Frame calculation is performed while control flow is still structured. This problem was solved by treating the structured control flow as unstructured by using ops to mark block boundaries.

```
kgen.func @coroutine_nested(%arg0: i1, %arg1: index, %arg3: index) async -> index {  
  %idx3 = index.constant 3  
  %result = kgen.call @foo(%idx3, %arg1) : (index,index) -> index  
  hlcf.loop "_loop_0" {  
    hlcf.if %arg0 {  
      hlcf.yield  
    } else {  
      %result4 = kgen.call @bar(%result, %arg3): (index,index) -> index  
      hlcf.loop "_loop_1" {  
        hlcf.if %arg0 {  
          hlcf.yield  
        } else {  
          co.suspend (%hdl) {  
            co.suspend.end  
          }  
          %result6 = kgen.call @foo(%result, %arg3): (index,index) -> index  
          hlcf.break "_loop_1"  
        }  
        hlcf.continue  
      }  
      %result6 = kgen.call @bat(%result, %arg3): (index,index) -> index  
      hlcf.break "_loop_0"  
    }  
    hlcf.continue  
  }  
  kgen.return %result : index  
}
```

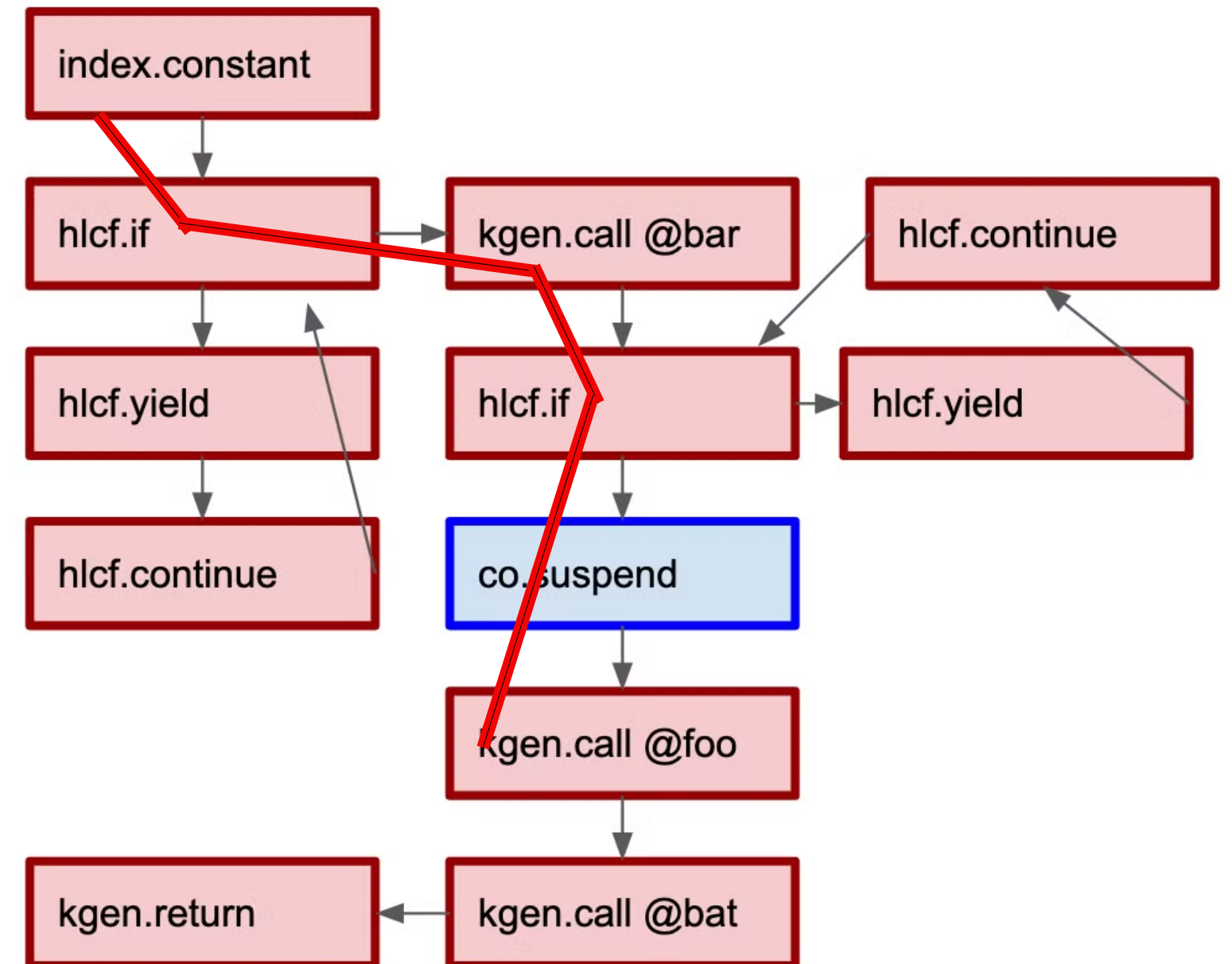


Frame Challenges



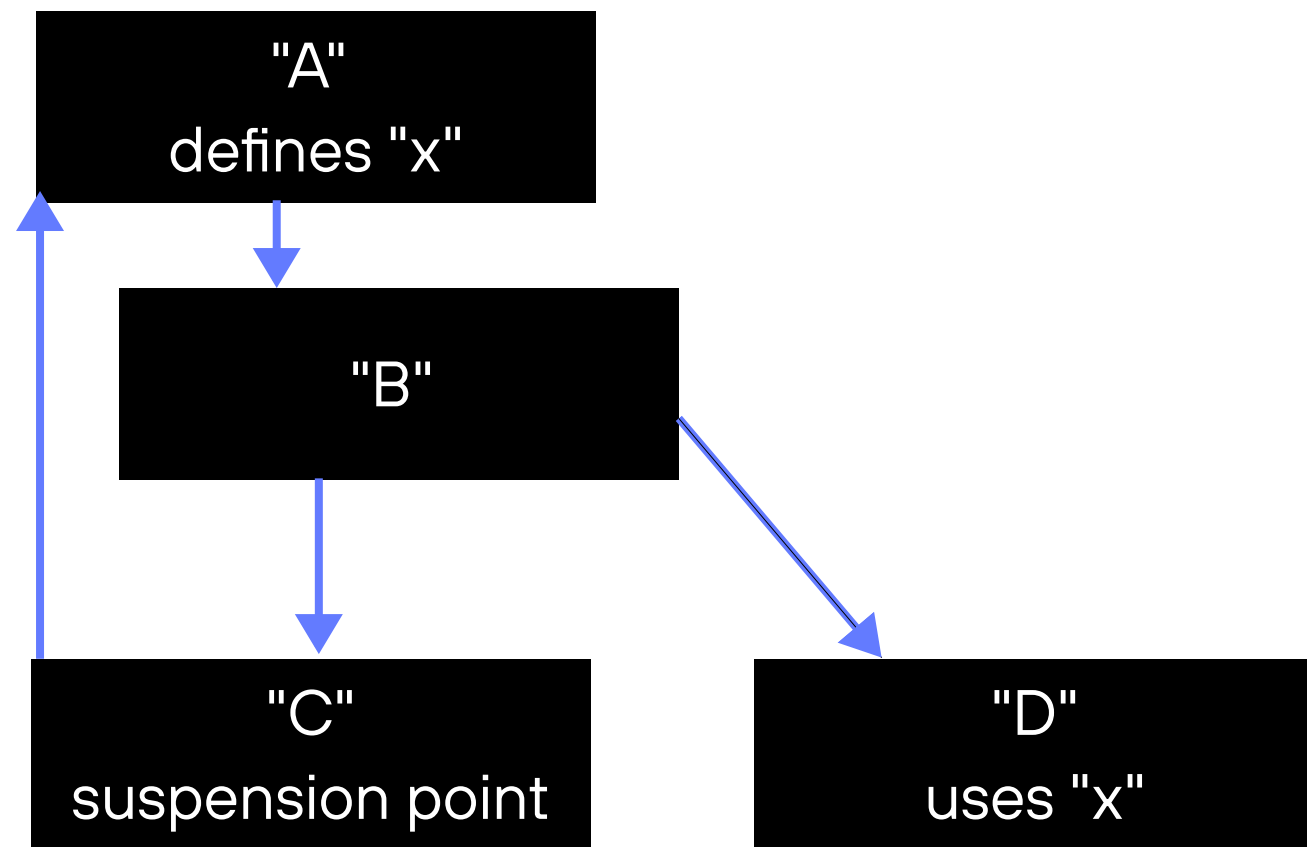
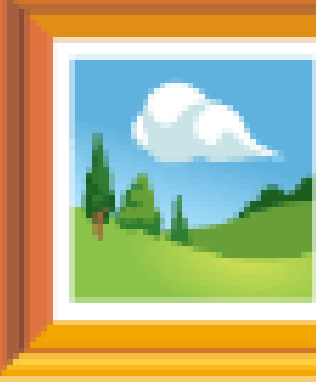
Frame calculation is performed while control flow is still structured. This problem was solved by treating the structured control flow as unstructured by using ops to mark block boundaries.

```
kgen.func @coroutine_nested(%arg0: i1, %arg1: index, %arg3: index) async -> index {  
  %idx3 = index.constant 3  
  %result = kgen.call @foo(%idx3, %arg1) : (index,index) -> index  
  hlc.loop "_loop_0" {  
    hlc.if %arg0 {  
      hlc.yield  
    } else {  
      %result4 = kgen.call @bar(%result, %arg3): (index,index) -> index  
      hlc.loop "_loop_1" {  
        hlc.if %arg0 {  
          hlc.yield  
        } else {  
          co.suspend (%hdl) {  
            co.suspend.end  
          }  
          %result6 = kgen.call @foo(%result, %arg3): (index,index) -> index  
          hlc.break "_loop_1"  
        }  
        hlc.continue  
      }  
      %result6 = kgen.call @bat(%result, %arg3): (index,index) -> index  
      hlc.break "_loop_0"  
    }  
    hlc.continue  
  }  
  kgen.return %result : index  
}
```



Frame Challenges

The algorithm to support whether or not a value needs to live on the frame

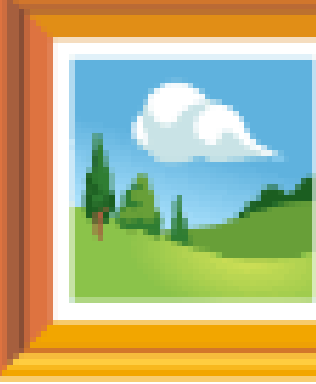


Unique paths with no cycles:

1. A, B, C → x is defined but never used
2. A, B, D → x is defined and used without passing suspension point

Path not considered: A, B, C, A, B, D → x is defined, function is suspended, then x is used! The conclusion is that "x" needs to live on the frame.

Frame Challenges

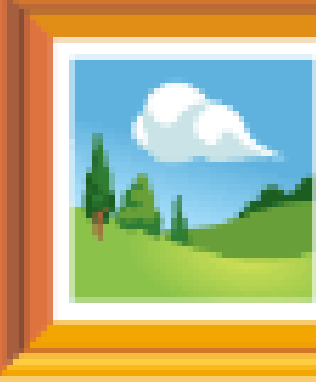


Cheap operations are cloned to avoid excessive space and loads and stores from frame. For example constants and pointer arithmetic are cloned at use sites if their definition is before a suspension point.

```
kgen.func @offset(%arg0: i1 borrow, %arg1: !kgen.pointer<none> byref_result) async no_inline {  
  %0 = pop.stack_allocation 2 x index marked  
  pop.stack_alloc.lifetime.start(%0) : !kgen.pointer<index>  
  %idx1 = index.constant 1  
  %1 = pop.offset %0[%idx1] : !kgen.pointer<index>  
  hlc.loop {  
    hlc.if %arg0 {  
      hlc.yield  
    } else {  
      hlc.break  
    }  
  }  
  co.suspend (%hdl) {  
    co.suspend.end  
  }  
  %4 = pop.load %1 : !kgen.pointer<index>  
  %3 = kgen.call @doSomething(%4) : (index) -> index  
  hlc.continue  
}  
%5 = pop.load %1 : !kgen.pointer<index>  
%6 = kgen.call @doSomething(%5) : (index) -> index  
pop.stack_alloc.lifetime.end(%0) : !kgen.pointer<index>  
kgen.return  
}
```

Don't put %1 on
the frame!

Frame Challenges

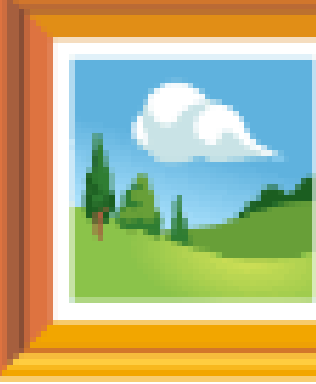


- Stack allocations must be replaced with frame allocations.
- stack allocations must rely on lifetime markers since their usage cannot be modeled through ssa value use def chains

```
kgen.func @coroutine(%arg1: index, %arg2: index) async -> index {  
  %0 = pop.stack_allocation 2 x index marked  
  pop.stack_alloc.lifetime.start(%0) : !kgen.pointer<index>  
  %idx1 = index.constant 1  
  pop.store %arg1, %0 : !kgen.pointer<index>  
  %1 = pop.offset %0[%idx1] : !kgen.pointer<index>  
  pop.store %arg2, %1 : !kgen.pointer<index>  
  %3 = kgen.call @use(%1) : (!kgen.pointer<index>) -> index  
  %22 = kgen.call @use(%0) : (!kgen.pointer<index>) -> index  
  co.suspend (%hdl) {  
    co.suspend.end  
  }  
  %2 = kgen.call @use(%1) : (!kgen.pointer<index>) -> index  
  pop.stack_alloc.lifetime.end(%0) : !kgen.pointer<index>  
  co.suspend (%hdl) {  
    co.suspend.end  
  }  
  %idx0 = index.constant 0  
  kgen.return %idx0 : index  
}
```

In this example, two index values are allocated on frame because of the usage across suspension points indicated by marker

Frame Challenges



Hot started coroutines have smaller frames. The resume function can still be used by both hot and cold started coroutines by storing first state values at the tail of the frame, which is stored at the tail of the coroutine

```
kgen.func @needsBoth(%arg0: index) async -> index {  
  %x = index.add %arg0, %arg0  
  %y = index.add %x, %x  
  co.suspend (%hdl) {  
    co.suspend.end  
  }  
  kgen.return %y : index  
}
```

Hot coroutine:

```
struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index)>
```

Cold coroutine:

```
struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>
```


Overview of Async Function Lowering

M

- For each async function, determine if it needs a hot ramp, a cold ramp, or both
- Preprocess the async function:
 - Move stack allocations as close to their first use as possible
 - Lower implicit suspension points
- Create coroutine for each asynchronous function
 - Calculate the frame (hot cold or both)
 - Generate the ramp function
 - Generate the resume function
- Replace hot invokes/cold invokes with calls to corresponding ramp functions
- Lower all other coroutine dialect ops. Now that we have the frame types, these are more or less GetElementPtr ops




```

module {
kgen.func @coroutine(%arg1: index) async -> index {
  %0 = kgen.call @foo(%arg1) : (index) -> index
  co.suspend (%hdl) {
    co.suspend.end
  }
  %final = index.add %0, %0
  kgen.return %final : index
}

kgen.func @trigger_creation(%arg1: index) async {
  %coro = co.invoke[(index) async -> index: @coroutine](%arg1)
  kgen.return
}
}

```

```

module {
  kgen.func @coroutine_ramp(%arg0: index) -> !kgen.pointer<struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>)>> {
    %idx72 = index.constant 72
    %idx8 = index.constant 8
    %0 = pop.aligned_alloc %idx8, %idx72 : <struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>>
    %1 = kgen.param.constant: i32 = <0>
    %2 = kgen.struct.gep %0[0] : <struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>>
    pop.store %1, %2 : !kgen.pointer<i32>
    %3 = kgen.struct.gep %0[1] : <struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>>
    %4 = kgen.create_closure[(!kgen.pointer<struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>>) -> (): @coroutine_resume]()
    %5 = pop.pointer.bitcast %4 : !kgen.signature<(!kgen.pointer<struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>>) -> ()> to !kgen.pointer<none>
    pop.store %5, %3 : !kgen.pointer<pointer<none>>
    %6 = kgen.struct.gep %0[8] : <struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>>
    pop.store %arg0, %6 : !kgen.pointer<index>
    %7 = pop.pointer.bitcast %0 : !kgen.pointer<struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>> to !kgen.pointer<struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>)>>
    kgen.return %7 : !kgen.pointer<struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>)>>
  }
  kgen.func @coroutine_resume(%arg0: !kgen.pointer<struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>>) attributes {coroutineType = !kgen.struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>} {
    %0 = kgen.struct.gep %arg0[8] : <struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>>
    %1 = pop.load %0 : !kgen.pointer<index>
    %2 = kgen.call @foo(%1) : (index) -> index
    %3 = kgen.struct.gep %arg0[7] : <struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>>
    pop.store %2, %3 : !kgen.pointer<index>
    %4 = kgen.param.constant: i32 = <1>
    %5 = kgen.struct.gep %arg0[0] : <struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>>
    pop.store %4, %5 : !kgen.pointer<i32>
    co.suspend {
      co.suspend.end
    }
    %6 = kgen.struct.gep %arg0[7] : <struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>>
    %7 = pop.load %6 : !kgen.pointer<index>
    %8 = index.add %7, %7
    %9 = kgen.struct.gep %arg0[6] : <struct<(i32, pointer<none>, (!kgen.pointer<none>) -> (), pointer<none>, pointer<none>, pointer<none>, struct<(index)>, index, index)>>
    %10 = kgen.struct.gep %9[0] : <struct<(index)>>
    pop.store %8, %10 : !kgen.pointer<index>
    kgen.return
  }
}

```


Omitted Details

M

- Lifetimes of values in coroutine frames as well as the lifetime of the coroutine itself
 - **Mojo has asap destruction**, alleviating the need for cleanup and destroy functions
- Error handling (we want to support throwing coroutines!)
- Debugging asynchronous functions
- Callbacks must be marked as musttail to avoid stack overflow in deeply nested coroutine chains
- If there are no suspension points, do not pay the price of asynchronous call
- What color is your function? Why do I have to mark a function with suspension points as "async"?

