High-Level Interface for Asynchronous I/O using C++20 Coroutines, io_uring, and eBPF

Hendrik Sieck

2022-03-17

Motivation: Event-Driven Programming

- System calls induce significant context switching costs
- Security vulnerability mitigations increase costs even more
- One approach: Lower number of system calls
- Aggregate system calls and push result processing into kernel
- Asynchronous interface leads to event-driven programming

Motivation: Event-Driven Programming

- System calls induce significant context switching costs
- Security vulnerability mitigations increase costs even more
- One approach: Lower number of system calls
- Aggregate system calls and push result processing into kernel
- Asynchronous interface leads to event-driven programming

Example using callbacks

```
function readFileToString(path, done) {
  open(path, function (file) {
    read(file, function (string) {
      close(function() {
        done(string);
      });
    });
  });
}
```

- Submission and completion split
- Does not scale well with complexity
- Hard to maintain and understand

Coroutines with async/await pattern

```
async function readFileToString(path) {
  const file = await open(path);
  const string = await read(file);
  await close();
  return string;
}
```

- Synchronous control flow
- Suspension points async. operations
- Synchronous code in between

Kernel Interfaces and C++20 Coroutines

io_uring

- Since Linux 5.1
- Asynchronous I/O
- Batching interface
- System call support

eBPF

- In-kernel virtual machine
- Attachable to events
- eBPF maps
- io_uring triggers

C++20 Coroutines

- Language support for coroutines
- Stackless coroutines
- Automatic state e.g. for variables
- Lowered to regular functions
- Allows common optimizations

Kernel Interfaces and C++20 Coroutines

TUHH Technische Universität Hamburg

io_uring

- Since Linux 5.1
- Asynchronous I/O
- Batching interface
- System call support

eBPF

- In-kernel virtual machine
- Attachable to events
- eBPF maps
- io_uring triggers

C++20 Coroutines

- Language support for coroutines
- Stackless coroutines
- Automatic state e.g. for variables
- Lowered to regular functions
- Allows common optimizations

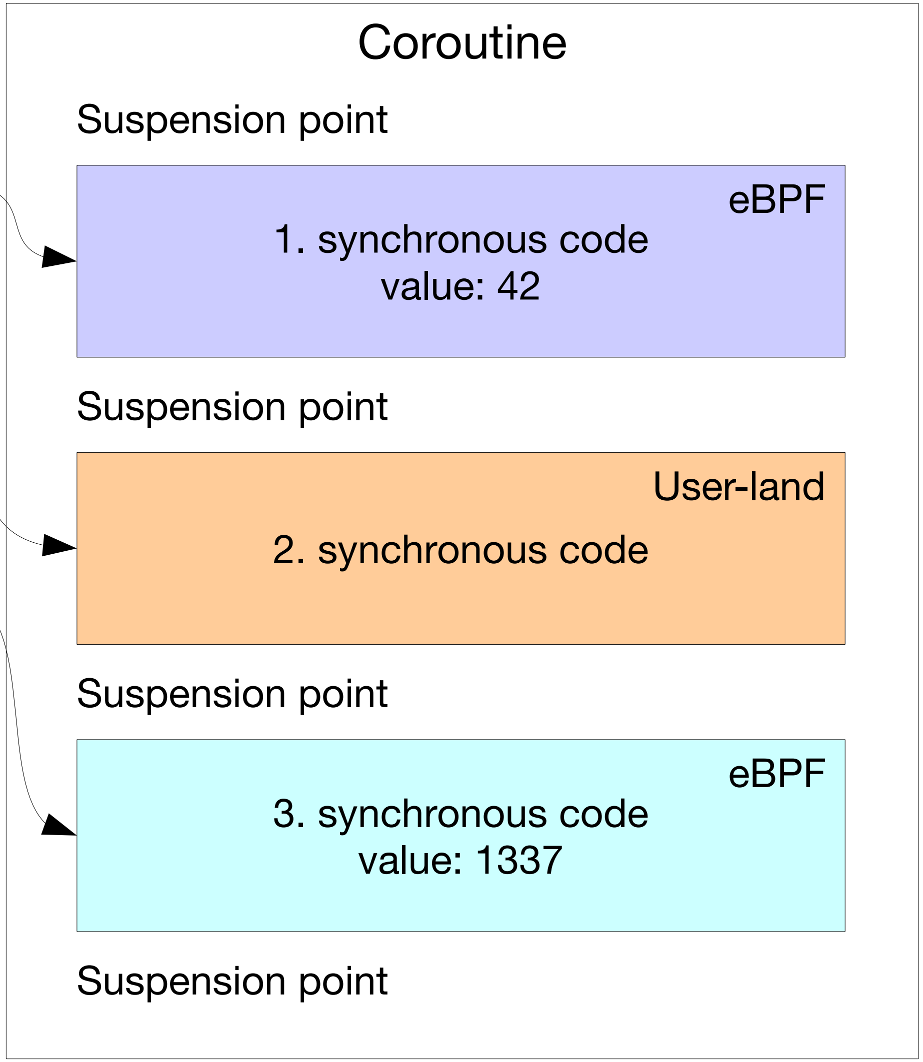
```
task coroutine(int value) {
  co_await resume_as_ebpf(true);
  std::cout << "1. synchronous code\n";
  std::cout << "value: " << value << "\n";
  co_await resume_as_ebpf(false);
  std::cout << "2. synchronous code\n";
  value = 1337;
  co_await resume_as_ebpf(true);
  std::cout << "3. synchronous code\n";
  std::cout << "value: " << value << "\n";
  co_await resume_as_ebpf(false);
}
```

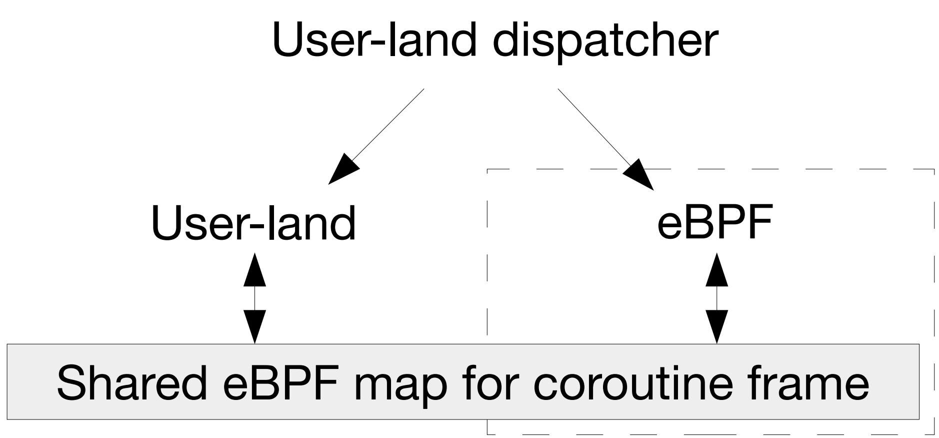
Contribution

- Selective dispatch to user-land or eBPF
  - Asynchronous dispatch via io_uring
  - Synchronous code sections in eBPF
- Self-contained executable

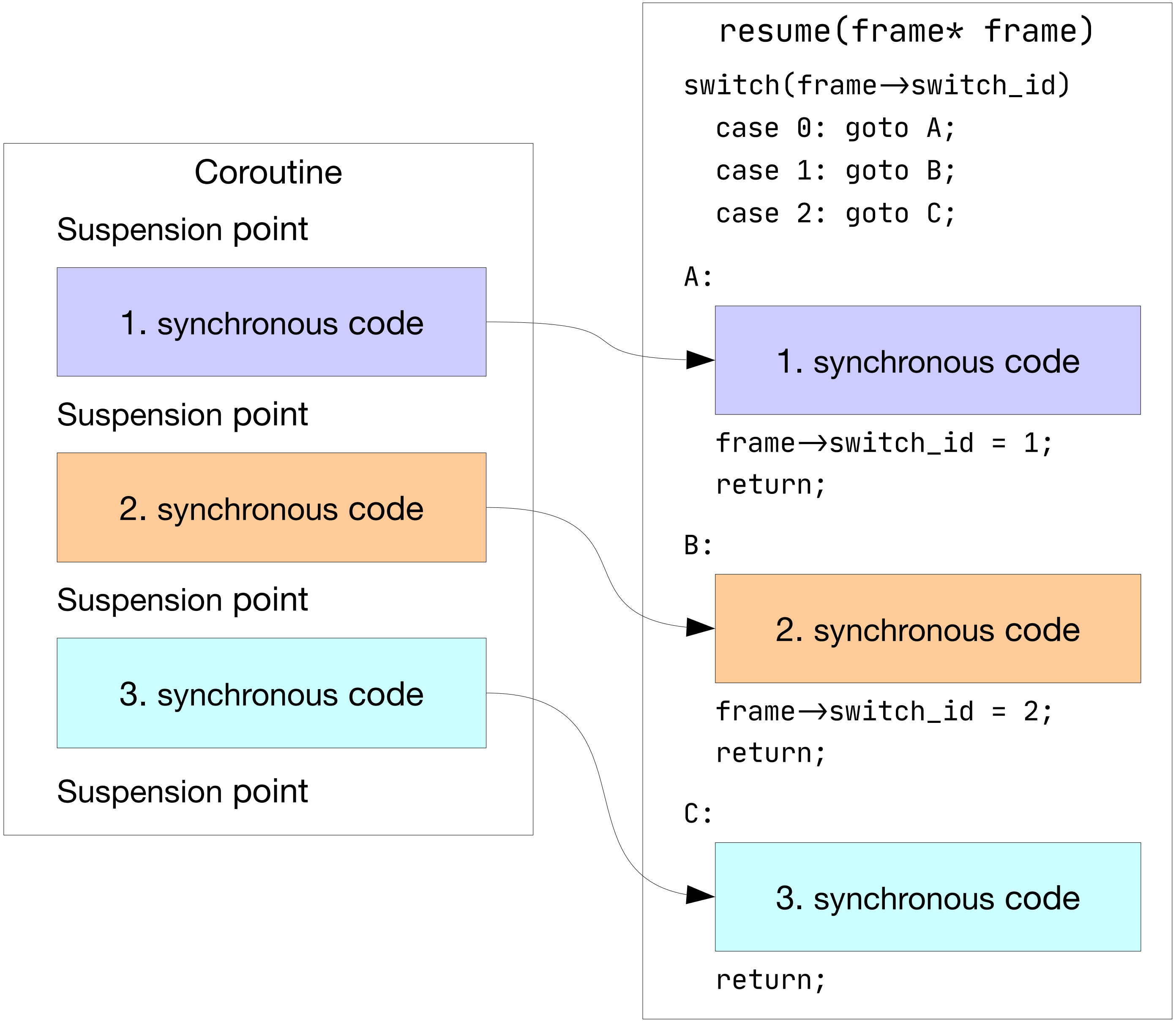
C++20 Coroutines Dispatching and Frame

```
task coroutine(int value) {
  co_await resume_as_ebpf(true);
  std::cout << "1. synchronous code\n";
  std::cout << "value: " << value << "\n";
  co_await resume_as_ebpf(false);
  std::cout << "2. synchronous code\n";
  value = 1337;
  co_await resume_as_ebpf(true);
  std::cout << "3. synchronous code\n";
  std::cout << "value: " << value << "\n";
  co_await resume_as_ebpf(false);
}
```





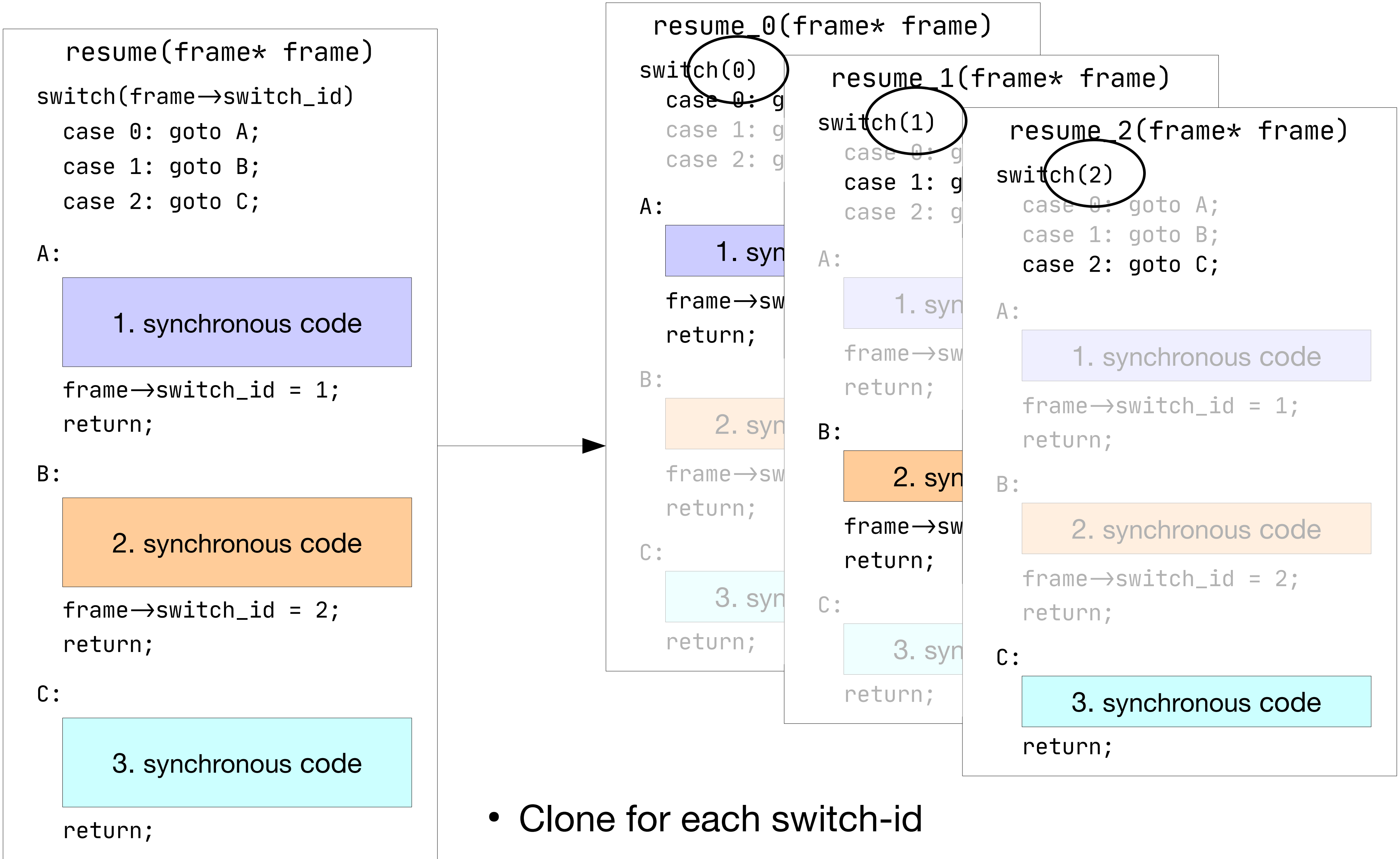
Switched-Resume Lowering in LLVM




```
struct frame {
  uint64_t switch_id;

  int value;
  // ... more local variables
}
```

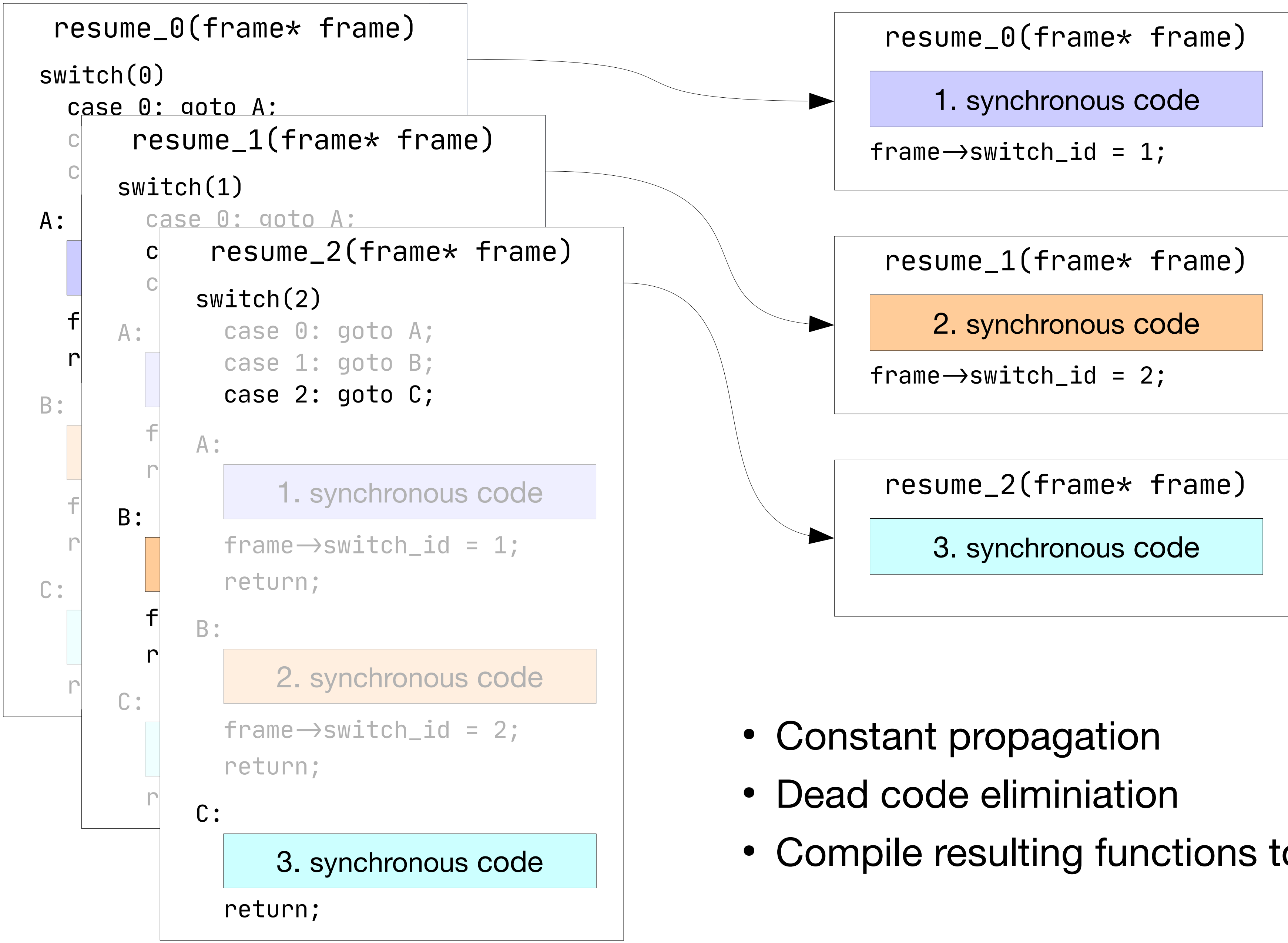
- Split coroutine
- Switch instruction
- Frame with switch-id

Resume Function Cloning



- Clone for each switch-id
- Replace switch-id with constant

Optimize Cloned Resume Functions



- Constant propagation
- Dead code eliminiation
- Compile resulting functions to eBPF

High-Level Interface

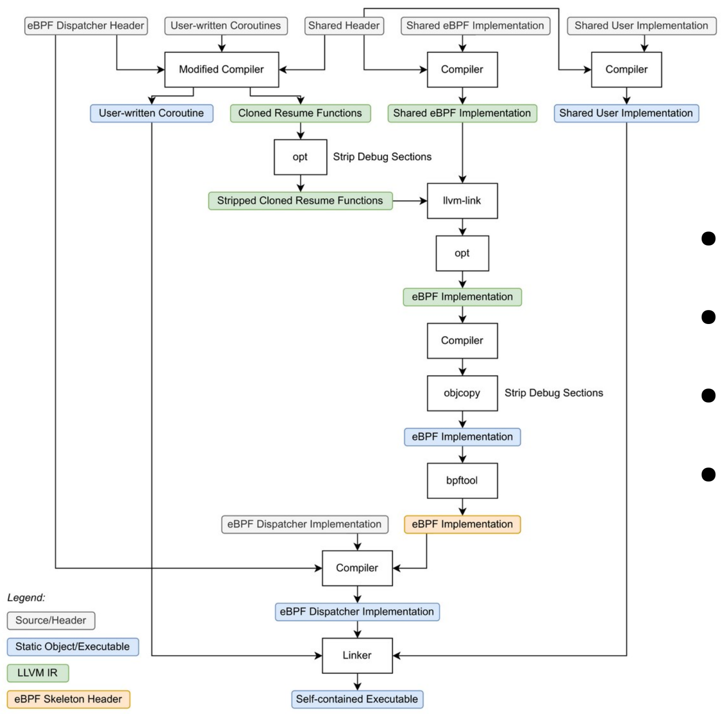
```
int main() {
  bpf_dispatcher__initialize();
  io_uring_service service;
  service.run(coroutine(service));
}

io_uring_task coroutine(io_uring_service &service) {
  co_await service.suspend(true);
  for (std::uint64_t i = 0; i < ITERATIONS; ++i) {
    io_uring__print_int(i);
    co_await service.suspend(true);
  }
  co_await service.suspend(false);
  io_uring__print_int(42);
}

struct io_uring_service {
  void run(io_uring_task task) {
    while (!task.coroutine.done()) {
      frame_extractor extractor{task.coroutine};
      if (task.coroutine.promise().resume_in_bpf) {
        bpf_dispatcher__dispatch(extractor.get_switch_id());
      } else {
        extractor.get_table()[extractor.get_switch_id()](extractor.get_frame());
      }
    }
  }
};
```

- Coroutine frame as eBPF map
- Boolean flag for dispatch target
- Shared functionality

Toolchain

Coroutine implementation

Shared functionality



- eBPF dispatcher
- Simple scheduler
- Modified Clang/LLVM
- bpftool

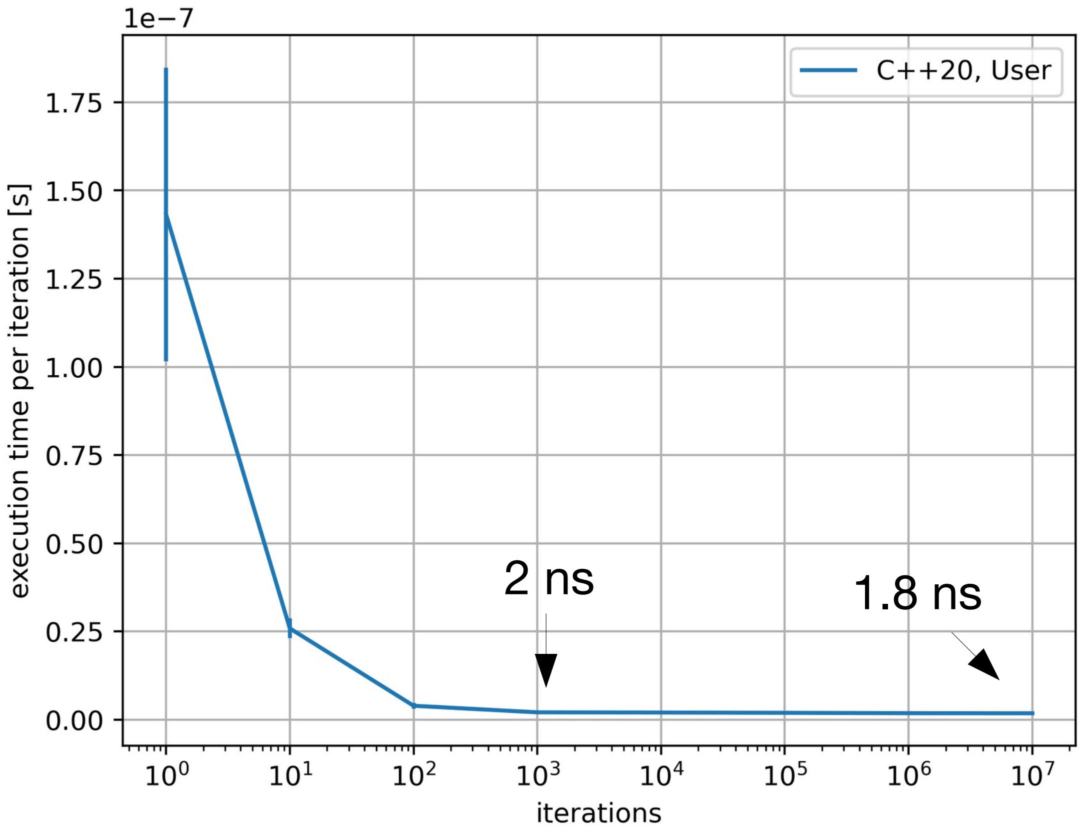
Self-contained executable → Patched kernel

with embedded eBPF programs

Evaluation: C++20 Coroutines without eBPF

How fast without any eBPF overheads?

```
io_uring_task coroutine(io_uring_service &service) {
  co_await service.suspend(false);
  for (std::uint64_t i = 0; i < ITERATIONS; ++i) {
    co_await service.suspend(false);
  }
}
```



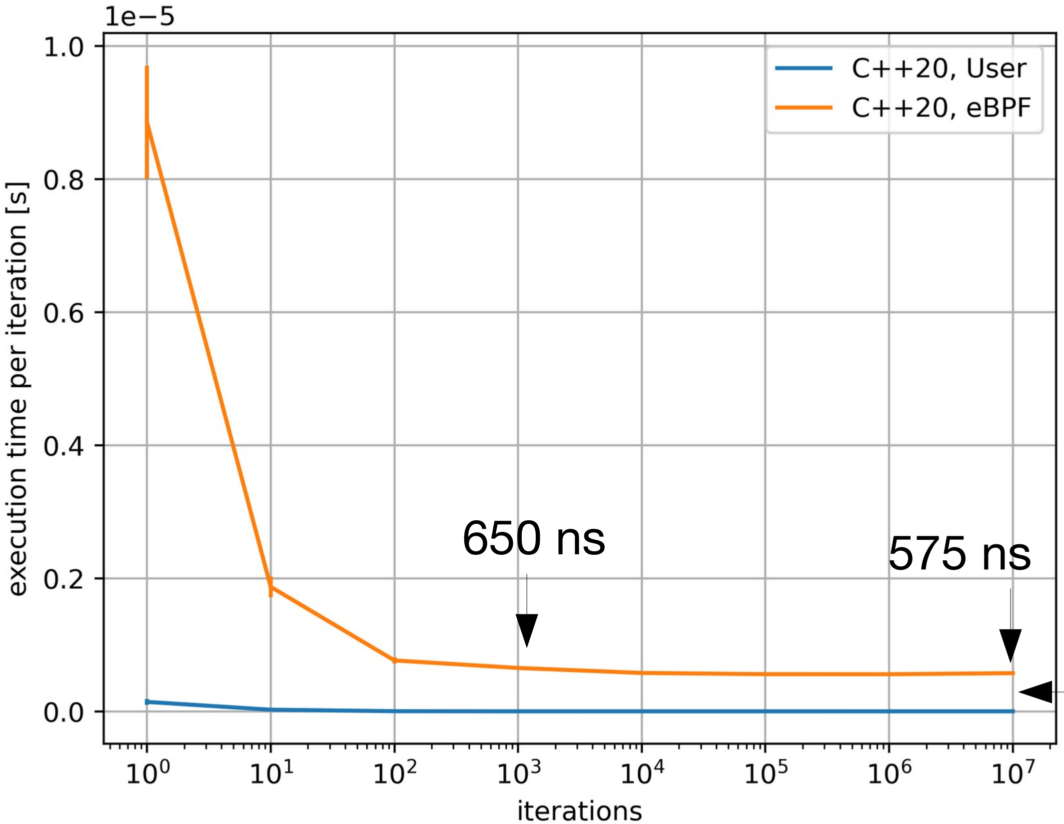
- Base case
- eBPF verification not included
- Constant overhead getting irrelevant
- Execution time scales proportionally

All benchmarks on Intel Core i7-4770 @ 3.4 GHz with 16 GiB RAM and Linux 5.14.
Means and standard deviations calculated over 100 samples.

Evaluation: C++20 Coroutines with eBPF

How fast is the proposed implementation?

```
io_uring_task coroutine(io_uring_service &service) {
  co_await service.suspend(true);
  for (std::uint64_t i = 0; i < ITERATIONS; ++i) {
    co_await service.suspend(true);
  }
}
```



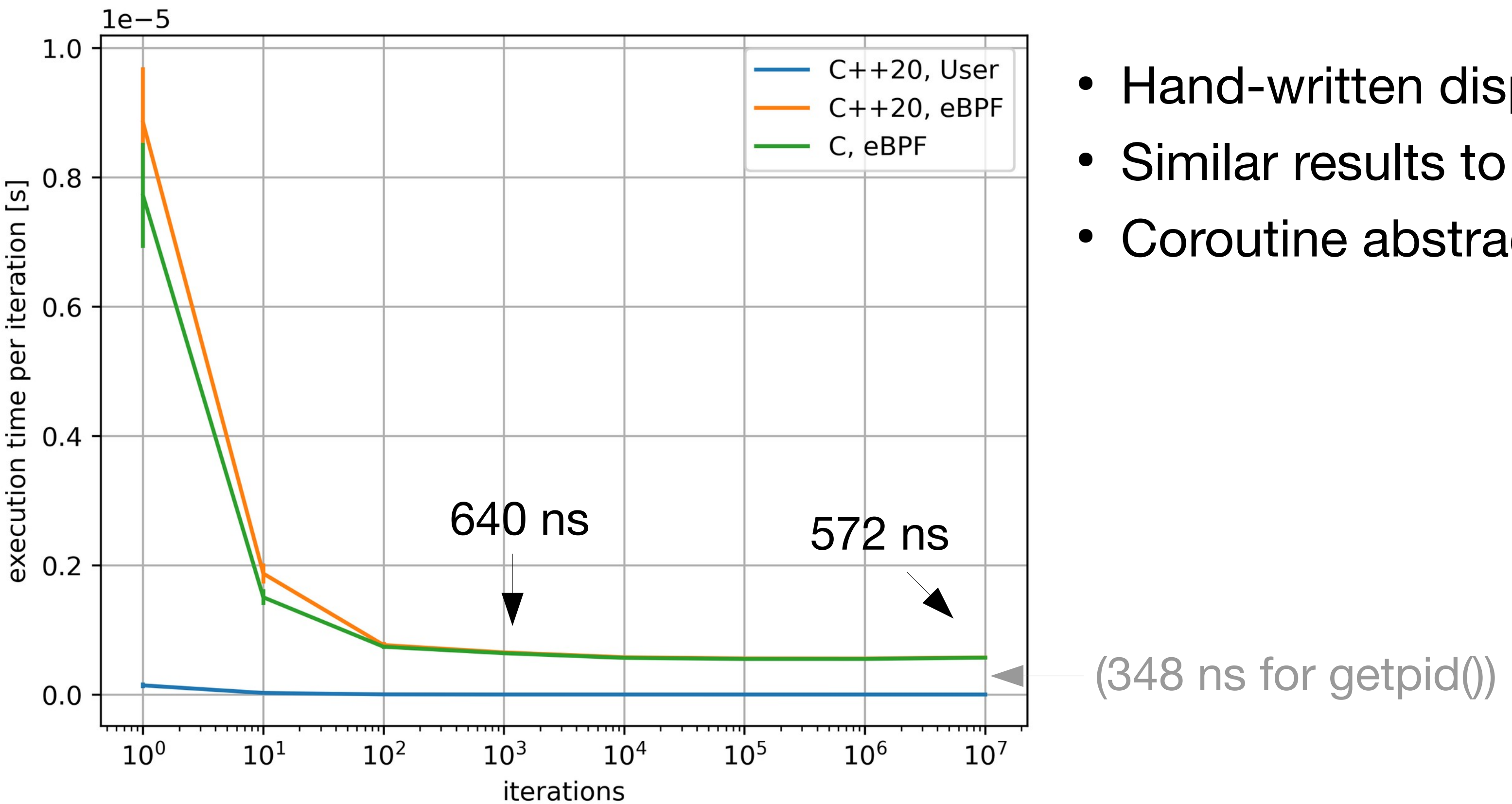
(348 ns for getpid())

- All resumptions as eBPF
- Constant overhead getting irrelevant
- Execution time scales proportionally
- In general slower (~350x)
- High overheads due to system calls
- io_uring and eBPF contribe less

Evaluation: C with eBPF

How much do coroutine abstractions cost?

```
SEC("iouring")
int loop_counter_incrementor(struct io_uring_bpf_ctx *context) {
  uint32_t index = 0;
  void *frame = bpf_map_lookup_elem(&context_map, &index);
  if (frame) {
    uint64_t *values = (uint64_t *)frame;
    values[0] += 1;
    values[1] = values[0] < ITERATIONS ? 1 : 2;
  }
  return 0;
}
```

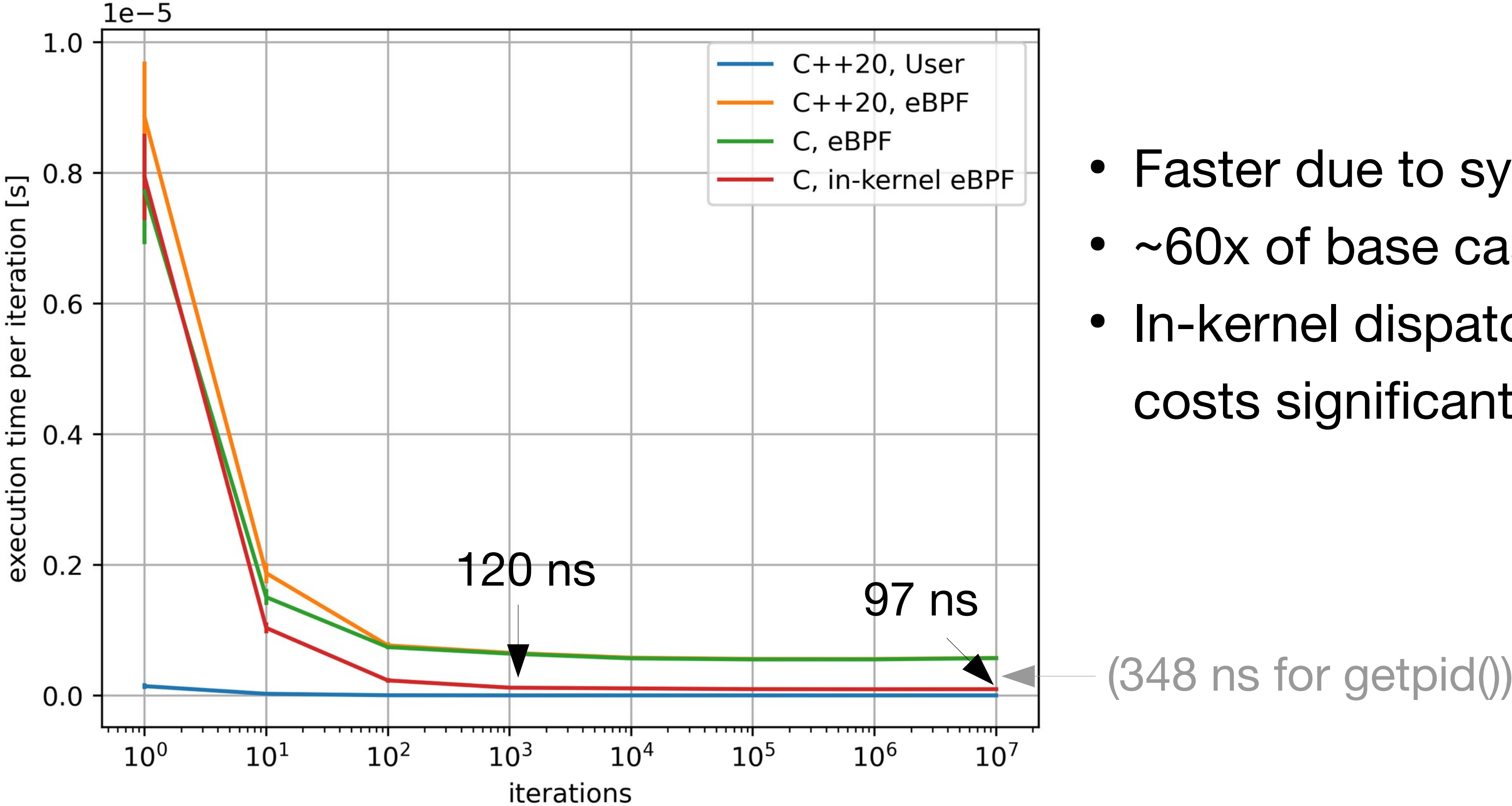


- Hand-written dispatcher
- Similar results to C++20, eBPF
- Coroutine abstraction has low costs

Evaluation: C with in-kernel eBPF

How fast without system call overheads?

- Dispatcher in eBPF program
- Single context switch for whole program



- Faster due to system call savings
- ~60x of base case
- In-kernel dispatcher decreases costs significantly

Conclusion

- Problem
  - Reducing system call overheads leads to event-driven programming
  - Complex control flow because of split implementation
- Proposed solution
  - Combine C++20 coroutines, io_uring, and eBPF
  - Selectively move synchronous code segments to eBPF
  - Self-contained executable with embedded eBPF programs
- Future work and outlook
  - In-kernel dispatcher saves costs
  - Add I/O operations
  - Exclude code or automatically detect if eBPF is compilable