Coroutines in BPF

Kumar Kartikeya Dwivedi Rishabh Iyer Sanidhya Kashyap

EPFL | Meta

Outline

Motivation

High-level Design

Safety

Implementation Notes

Next Steps

Motivation

- A lot of patterns where a logical task is composed across “space” and “time”.
  - I.e., Across various execution contexts, and interspersed with delays.
- Many existing programming patterns in BPF map to this idea.
  - Work deferral primitives (timers, wq, task_work, etc.).
- Many existing and speculative use cases can be unblocked through such a primitive.
  - In general, for controlling scheduling behavior of kernel computation.
- Requests are unit of concurrency for applications.
  - Task execution that encompasses the kernel and user space.

Example 1: Stack trace collection

- Collect kernel + user stack traces on NMI events.
- We introduced task_work to defer user stack trace collection on the return to user mode path from the perf NMI program.
- Represent the entire program synchronously with a suspension point to wait until task work scheduling.

```
Task<int> perf_event(void *ctx) {
        collect_kernel_stack_trace(ctx);
        co_await task_work_schedule();
        collect_user_stack_trace(ctx);
        co_return emit_event(ctx);
}
```

Example 2: Defer qdisc_run()

- Tx path for the application ends up hitting qdisc_run().
  - sendmsg() -> tcp_write_xmit() -> IP layer -> qdisc -> qdisc_run().
- Well-understood source of latencies.
  - We might end up doing work for packets queued by other applications.
  - Especially bad under overcommit of CPUs where execution CPUs may overlap.
- Deferral from application core and controlling how much CPU bandwidth is allotted to work.
- Also requires control over threads, how queues containing coroutines are drained, CPU scheduling, etc.

Example 3: Application concurrency

- Some academic use cases implement applications as BPF programs.
- E.g., KV-store in BPF, but based on request service times, the XDP program ends up hogging the CPU.
- Offer ability to precisely control computation and its scheduling behavior / CPU bandwidth.
- Not very realistic, but illustrates and exercises offered flexibility.

History

- The idea of a hook, with a well-defined interface for interaction, and produced output is deeply entrenched (in BPF, and things that came before it).
- Inversion of control. You can do everything with callbacks that you do with continuations.
- Most of the benefits come from the ability to reason about the code locally, which becomes harder when making control flow unstructured.
- Academic curiosity: what would it take to theoretically apply continuations to this model?
- Surprisingly, this has not been done in context of kernel extensions in the past.

Separation of Concerns

- Modularize and separate parts of an extension responsible for orthogonal steps.
  - Scheduling of work, from the actual business logic that needs to be scheduled, instead of mixing both together.
  - Verify programs that perform computation and schedule independently, in isolation.
  - Enable local reasoning. Scale horizontally.
- Prove more complex properties and stack stronger properties on top of the kernel verifier, which only cares about maintaining kernel's integrity.
- E.g., a functional property of how I schedule work using this primitive may affect overall system availability (whcih encompasses the kernel + application).

High-level Design

Design

- Stackless coroutines (no save/restore, state across suspension lives in a separate frame).
  - Equivalent to (indirect) function call on every resumption.
- Leverage the compiler to split the coroutine body and set up the coroutine frame.
  - Pros: Kick in compiler optimizations, allocation elision, suspension elision.
  - Cons: The verifier needs to understand each distinct lowering conventions. (C++, Rust).
- The verifier ensures the observed control flow does not violate the kernel's safety invariants.
- The BPF runtime provides primitives and preserves invariants the coroutine code relies upon.
  - Allocation of the coroutine frame, precise tracking (like a stack) in the verifier.
  - Access exclusivity invariant for the coroutine frame (across suspension points).

C++ Coroutines

- Switch-resume lowering.
- Resume is the resumption handler, destroy is used to free a suspended coroutine.
- An index in the activation frame keeps track of the current position in the corutine body.
- Every resumption goes through the switch statement, and uses the index to decide where to jump to.

```
struct frame {
    void (*resume)(void *handle);
    void (*destroy)(void *handle);
    u8 index;
    ....
};
```

```
void resume(void *handle) {
    switch (((struct frame *)handle)->index) {
        case 0: goto x;
        case 1: goto y;
        ...
    }
```

C++ Coroutines

- The program implements a coroutine type (e.g., Task<T>), when a function uses co_await, co_yield, co_return, compiler calls methods into the class following a specific contract

```
Task<int> foo(void) {
        xyz();
        co_await schedule_user_mode();
        abc();
        co_await schedule_timer(5ns);
        ...
        co_return 0;
}
```

```
Tash<int> foo(void) {
        xyz();
        if (Awaitable{}.await_ready()) {
                await_suspend(handle);
                await_resume();
        }
```

```
allocate coroutine_state;
construct promise;
auto ret = promise.get_return_object();

co_await promise.initial_suspend();

try {
    auto awaiter = make_awaiter(g());

    if (!awaiter.await_ready()) {
        save_resume_point();
        std::coroutine_handle<promise_type> h =
            std::coroutine_handle<promise_type>::from_promise(promise);

        awaiter.await_suspend(h);
        return_to_caller_or_resumer();
    }

resume_point:
    int x = awaiter.await_resume();
    promise.return_value(x + 1);
}
catch (...) {
    promise.unhandled_exception();
}

co_await promise.final_suspend();
```

---

1. Heap allocation of the coroutine frame — `promise_type::operator new` lowered

`fengshui_coro_tp+0x8 .. +0x40`:

```
1: r1 = 0x40                          ; sizeof(coroutine frame), chosen by LLVM
2: r2 = 0x0                           ; second arg (ctx) — nullptr in this overload
3: call bpf_coro_frame_alloc          ; R_BPF_64_32 → kfunc
4: r6 = r0                            ; r6 = frame ptr (handle)
5: if r6 != 0x0 goto +0x3 <ok>
6: r1 = 0x0
7: call bpf_throw                     ; alloc failed → abort
8: call __bpf_trap
```

This is the entire `static void *operator new(size, void *ctx)` body, called by the compiler-emitted ramp.

---

2. Frame initialization — resume/destroy fn pointers + state index

`fengshui_coro_tp+0x48 .. +0x80`:

```
9:  w8 = 0x0
10: *(u8  *)(r6 + 0x38) = w8          ; resume_index = 0  (initial_suspend point)
12: r1 = 0x178 ll  ; reloc → .text     ; destroy fn = .text + 0x178
14: *(u64 *)(r6 + 0x8)  = r1
15: r1 = 0x0   ll  ; reloc → .text     ; resume  fn = .text + 0
17: *(u64 *)(r6 + 0x0)  = r1
```

Layout LLVM picked:
- `+0x00` resume fn ptr
- `+0x08` destroy fn ptr
- `+0x10..0x37` spill slots for live state across the suspend
- `+0x38` one-byte resume index (the state machine's `switch` selector)

---

3. State-machine dispatch — top of `bpf_main.resume`

`.text+0x0 .. +0x10`:

```
0: r6 = r1                            ; r1 = frame (handle) on entry
1: w1 = *(u8 *)(r6 + 0x38)            ; load resume_index
2: if w1 != 0x0 goto +0x11            ; index==1 → jump to post-suspend block
                                      ; fall through = index==0 (initial entry)
```

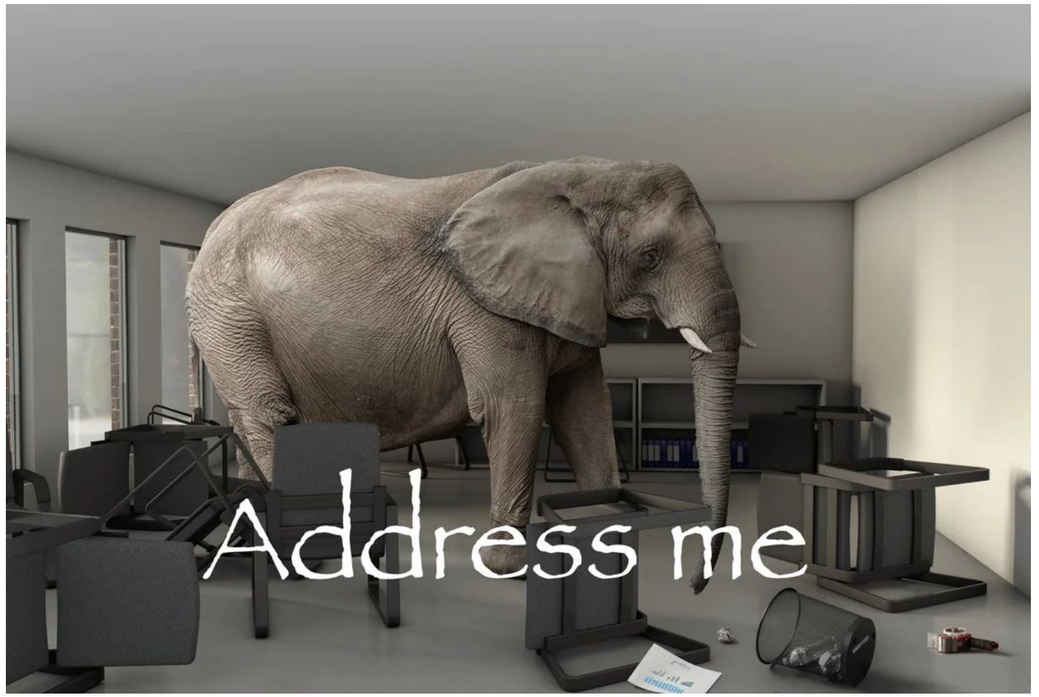
Verification approach

- Everything boils down to BPF assembly.
- We need to ensure that the program manipulates the frame correctly.
  - Resume, destroy, and state are things whose offset have to be known to the verifier.
  - Resume, destroy are always the first 16-bytes.
  - State offset needs to be passed (through BTF).
- Explore destroy path on every suspension of the coroutine.
  - The compiler generates all code to free held resources (assuming we're using RAII), and then calls delete on the coroutine frame (which calls the BPF-specific kfunc to release it).
- Paths to explore on resumption depend on precision of state of the coroutine frame.
  - If we know the current state index, we will only explore the specific continuation.

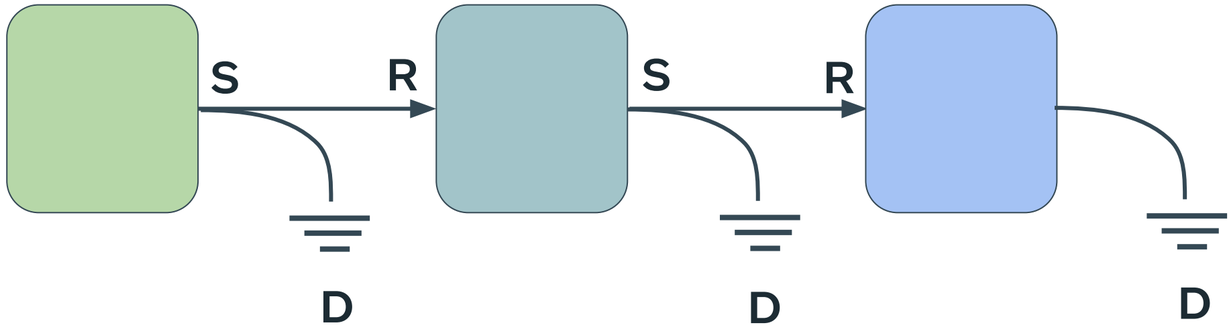
Continuation passing

- Stashing delimited continuations / suspended coroutines enables more control over scheduling properties.
  - Placing suspended computations in queues allows us to control sojourn time, and differentiate between work.
- How do we make sure the remainder runs in the right execution context?
- When we draw a handle from a map / queue, how do we reason about the state of the coroutine frame?
- How do we reason about the paths we need to explore when resuming such a coroutine.

Safety



Safety



- How do we reason about safety in a principled way?
- Consider each synchronous segment as its own run-to-completion extension.
- For each segment, we understand and enforce safety in isolation already.
  - There is a fixed execution context.
  - Protection scope for objects (RCU or RCU tasks trace), etc.
- We introduce three new components to the picture.
  - The coroutine frame, its lifetime, and the lifetime of objects within it.
  - New control flow after every suspension (S) point (either resumption (R) or destruction (D)).
  - Separation in execution (in both space / time).
    - Execution may change context for each segment.
    - Execution may be separated in time, controlled by the user.

Coroutine Frame

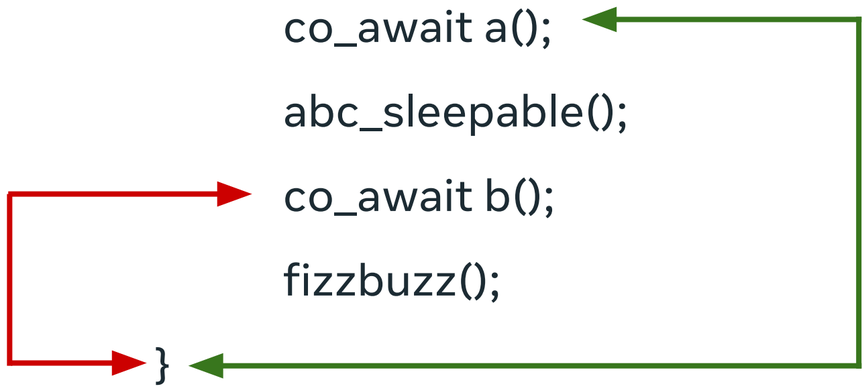
- Ensure safety properties for the frame apply, recursively.
  - E.g. Map values that do not survive segment execution get invalidated.
    - We may want to use SRCU to address this.
  - E.g. Refcounts stay valid.
- Invariant: A coroutine frame is only mutated by the coroutine body.
  - The kernel will ensure this as a guarantee, also exploit this to specialize and elide allocations.
    - E.g. reusing packet frames to house coroutine frames when ctx == skb / xdp, etc.

Spatial Safety (Execution Context)

- We explore resumption and destruction (resume(), destroy()) on every suspension.
  - Summarize “effects” that matter for safety.
  - E.g. Sleepable, TakesLocks, InvalidatesPacket, etc.
  - Imbue them in the type of the handle (e.g., when stashed in a map).
- When the handle is stashed, the type will carry information about constraints for the remaining delimited continuation (in terms of effects).
- When the handle is recovered, the type carries the constraints that need to be check on resumption. E.g. whether the execution context is safe for resumption.
- Effect types are expressive with their own algebra, to allow us to map safety properties.
  - E.g. sleepable handles can have non-sleepable handles stashed into their slot.
    - We will be conservative when enforcing execution context checks around resumption.

Spatial Safety (Execution Context)

```
Task<int> foo(void) {
        xyz();
        co_await a();
        abc_sleepable();
        co_await b();
        fizzbuzz();
}
```



Green: Sleepable

Red: Non-sleepable

More importantly, the Green continuation includes the Red continuation.

Temporal Safety (Time)

- Limit ourselves to a coarse safety floor.
  - The kernel should not become *unavailable*.
- Not crashing the kernel is a subset of this core property.
- Disallowing accumulation of coroutines in queues until the system runs out of memory.
  - Charge to the owner and release coroutines if queues are not polled in time.
- Disallowing liveness or starvation.
  - Poll queues to ensure they are drawn in a bounded amount of time, asynchronously.
- All complex verification for more bespoke properties should be layered on top.
  - Fairness to enforce certain QoS for different work.
- Core idea: Verifier should permit most flexibility while upholding the core invariant. All other restrictions to trade flexibility for safety are through verification layers on top.

Temporal Safety (Time)

- More extensive verification pushed in user space.
- Ensure that the way programs are constructed is *conducive* to such stacking of verification.
  - The scheduler that operates on abstract computation is isolated and reasoned about locally.
  - Queues are explicit, and have well-defined semantics (sequences).
  - Basic temporal properties through local reasoning (e.g., bounded steps visiting all queues, etc.).
- We can summarize more effects for computation and tie it to handle types which are only consulted by the user space verification layer to enforce various invariants.
- The kernel ensures worst-case behavior is well-behaved and allows recoverability, the rest is optional.
- Opens up avenues for more exploration and experimentation.

Implementation Notes

Verifier / Runtime

- Track the state of the coroutine frame the same way the stack is tracked.
  - Same logic for stack tracking, same implications for verification (pruning, precision, etc.).
- bpf_coro_frame_alloc()/bpf_coro_frame_alloc_free() backed by kmalloc_nolock() for allocating and releasing coroutine frames.
- Enable indirect calls.
  - Invocation of the resume function from a coroutine frame.
- Enable C++ tokens in keyword identifiers (<, >, ::, etc.).
- Imbue effect types into the handle type when stashed in a map / queue.
- Polling queues for liveness (at runtime).
  - O(n) queues per program x O(n) programs.

Compiler / BTF

- Most heavy lifting already done for C.
- Enable freestanding C++ target for BPF.
  - No RTTI, no exceptions.
  - Expose BPF C attributes to C++.
- Enable aggregate return types (limited to 8 bytes).
  - std::coroutine_handle<>, bpf::Task<T>, etc.
  - Bigger return values will require generic aggregate return support.
- Set config macros to reuse standard headers (like <coroutine>), provide shims for others (<new>, etc.).

Code Blame 35 lines (33 loc) · 1.2 KB Raw

```
// SPDX-License-Identifier: GPL-2.0
// BPF freestanding libc++ configuration.
// Disables threads, filesystem, locale, RTTI, and hardening.
#ifndef _LIBCPP___CONFIG_SITE
#define _LIBCPP___CONFIG_SITE

#define _LIBCPP_ABI_VERSION 1
#define _LIBCPP_ABI_NAMESPACE __1
#define _LIBCPP_ABI_FORCE_ITANIUM 1
#define _LIBCPP_ABI_FORCE_MICROSOFT 0
#define _LIBCPP_HAS_THREADS 0
#define _LIBCPP_HAS_MONOTONIC_CLOCK 0
#define _LIBCPP_HAS_TERMINAL 0
#define _LIBCPP_HAS_MUSL_LIBC 0
#define _LIBCPP_HAS_THREAD_API_PTHREAD 0
#define _LIBCPP_HAS_THREAD_API_EXTERNAL 0
#define _LIBCPP_HAS_THREAD_API_WIN32 0
#define _LIBCPP_HAS_THREAD_API_C11 0
#define _LIBCPP_HAS_VENDOR_AVAILABILITY_ANNOTATIONS 0
#define _LIBCPP_HAS_FILESYSTEM 0
#define _LIBCPP_HAS_RANDOM_DEVICE 0
#define _LIBCPP_HAS_LOCALIZATION 0
#define _LIBCPP_HAS_UNICODE 0
#define _LIBCPP_HAS_WIDE_CHARACTERS 0
#define _LIBCPP_HAS_TIME_ZONE_DATABASE 0
#define _LIBCPP_INSTRUMENTED_WITH_ASAN 0
// _LIBCPP_HARDENING_MODE_NONE = (1 << 1)
#define _LIBCPP_HARDENING_MODE_DEFAULT (1 << 1)
// _LIBCPP_ASSERTION_SEMANTIC_IGNORE = (1 << 2)
#define _LIBCPP_ASSERTION_SEMANTIC_DEFAULT (1 << 2)
#define _LIBCPP_LIBC_PICOLIBC 0
#define _LIBCPP_LIBC_NEWLIB 0
#define _LIBCPP_LIBC_LLVM_LIBC 0

#endif
```

Next Steps

Rust

- Rust uses a slightly different lowering.
  - No resume function in the coroutine frame.
  - Still preserves a state discriminant.
- We must ensure the resume function for a given frame type cannot be invoked on other frames.
- The rest should just follow from what was done for C++, and what the verifier already explores.
- Should be doable, but the verifier treats this lowering ABI slightly differently from the C++ one.

Unifying user space and kernel

- Share continuation frame across user space and kernel coroutines.
  - A request that encompasses the kernel and user space can be represented as a single language level task.
- The compiler splits the body and compiles them with different backends.
- The frame essentially gets treated as an arena, since the contents cannot be trusted anymore.
- The only bottleneck is trusted kernel data.
  - Trusted pointers saved in frame, precise scalars verifier thinks it knows the value of, etc.
- Possible solutions:
  - Declare bankruptcy (but then it’s not very useful).
  - Use a shadow frame and duplicate writes of precise state to shared and shadow frame.
- Needs more thought.


Meta