eBPF-specialized Kernel for I/O Intensive Applications

Kumar Kartikeya Dwivedi, Rishabh Iyer, Sanidhya Kashyap

EPFL

THE UNIVERSITY OF CALIFORNIA · BERKELEY · 1868

Before we begin…

- Any misrepresentation of other work is my mistake / responsibility.
- I’m looking for feedback, these are early ideas.
- Please poke holes!

Problem Statement

- Growing hardware capacity and speed is highlighting host CPU bottlenecks.
    - CPU speeds not growing as quickly.

Problem Statement

- Growing hardware capacity and speed is highlighting host CPU bottlenecks.
  - CPU speeds not growing as quickly.

- Operating systems overfit for applications due to genericity.
  - E.g. much of the functionality in the data path may not be needed, but cost is paid.

Problem Statement

- Growing hardware capacity and speed is highlighting host CPU bottlenecks.
    - CPU speeds not growing as quickly.

- Operating systems overfit for applications due to genericity.
    - E.g. much of the functionality in the data path may not be needed, but cost is paid.

- Solutions are too specialized / disruptive.
    - Kernel-bypass I/O stacks: no multi-tenancy, workload isolation.
    - Dataplane OSs, require re-architecting applications: poor compatibility, too costly.

Three Scenarios

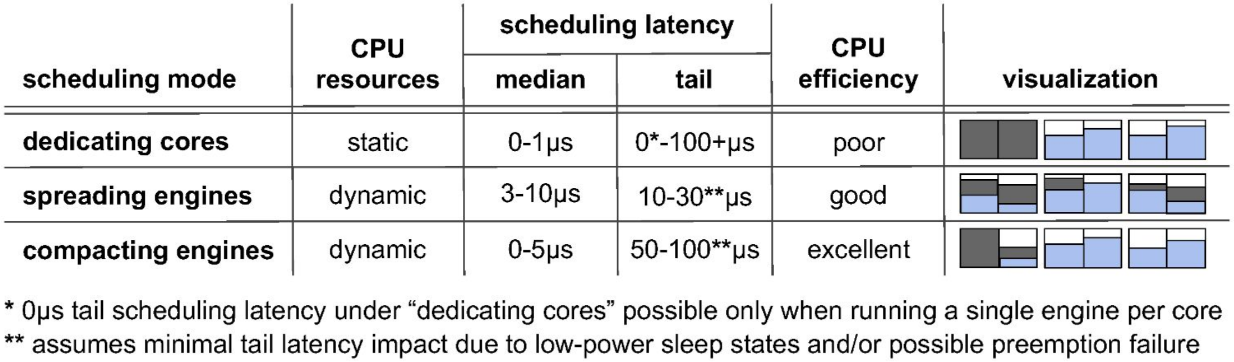
- Latency vs Efficiency
- Isolation
- Dataplane OS

Latency vs Efficiency: Snap

- User space networking stack.
  - Feature velocity (not our focus).
  - Navigates lower latency vs better efficiency.
- ‘Engine’ threads handle packet processing.
- Three scheduling models: Dedicated, Spreading, Compacting.

Snap

- Dedicated: Busy-polling pinned engine thread per-core, no co-location.
- Spreading: Spread work to available idle cores, driven by interrupts.
- Compaction: Spreading for gaining capacity; use queuing delay for SLO compliance, and densely pack bin-pack work to free capacity.
    - Hybrid optimistic polling + interrupt driven notifications.



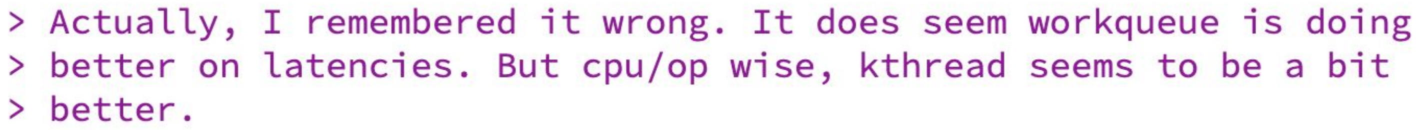
scheduling mode	CPU resources	scheduling latency		CPU efficiency	visualization
		median	tail		
dedicating cores	static	0-1µs	0*-100+µs	poor	
spreading engines	dynamic	3-10µs	10-30**µs	good	
compacting engines	dynamic	0-5µs	50-100**µs	excellent	

* 0µs tail scheduling latency under “dedicating cores” possible only when running a single engine per core

** assumes minimal tail latency impact due to low-power sleep states and/or possible preemption failure

Latency vs Efficiency: TAPI

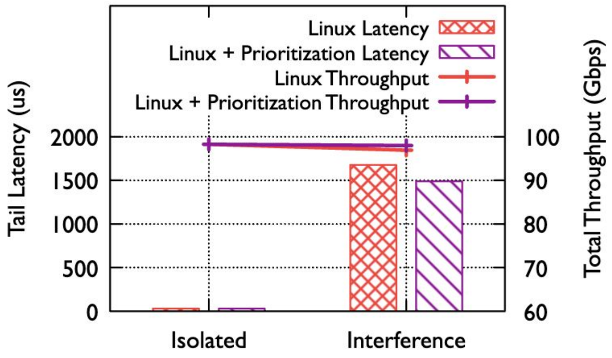
- Jakub’s proposal to realize what Snap’s compacting engines mode do, but for Linux’s netstack.
- Use of work queues as the execution context; wq items as the unit of concurrency.
- 3 pinned NAPI kthreads at 30% CPU util. vs 1 wq kthread at 90% CPU util.
- Avoid millisecond-scale latency spikes (say when co-locating NAPI kthreads).



```
> Actually, I remembered it wrong. It does seem workqueue is doing
> better on latencies. But cpu/op wise, kthread seems to be a bit
> better.
```

Isolation

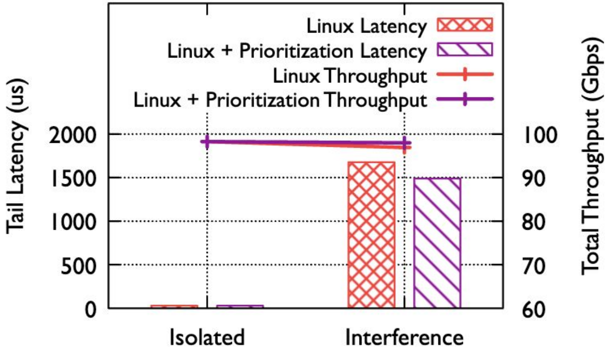
- Figure from NetChannel (Fig 4).
- 8 cores in the same NUMA node.
- 1 latency critical thread doing networking, 8 batch threads doing networking.



(a) P99.9 latency (μs) and total throughput (Gbps)

Isolation

- Since threads > cores, L-app may share core with T-app.
- Network stack processing of L-app may be queued behind T-app.
- 37x tail latency inflation.

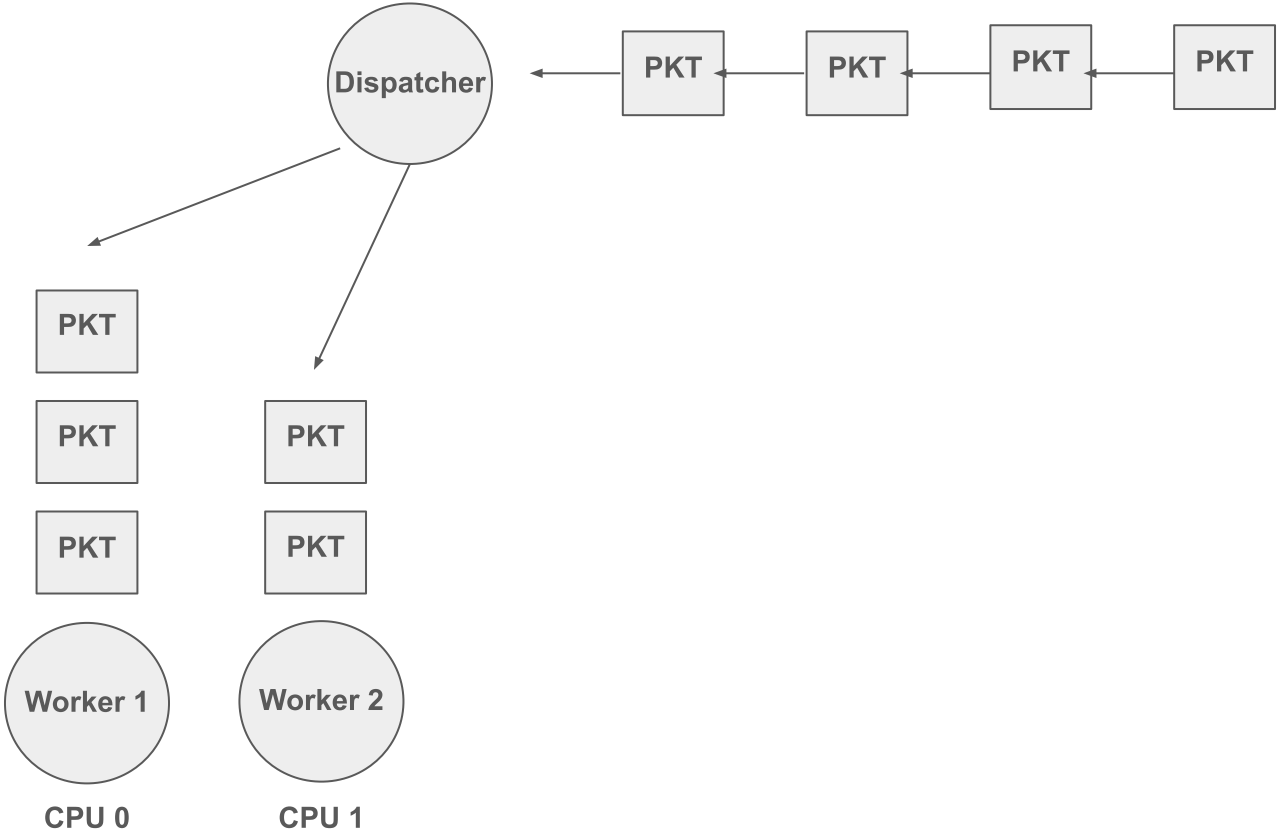


(a) P99.9 latency (μs) and total throughput (Gbps)

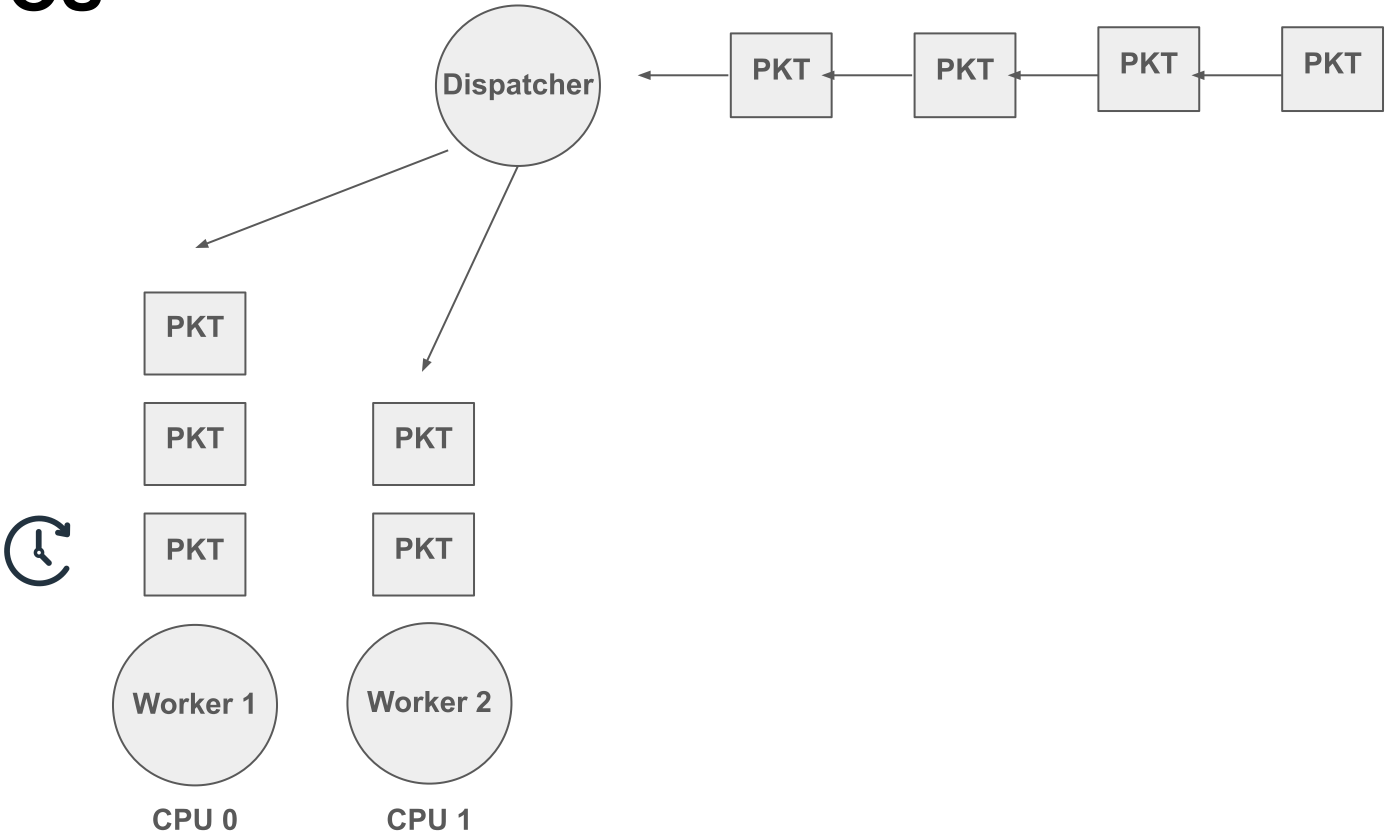
Dataplane OS

- One central “dispatcher” steering all packets onto “worker” cores.
- Worker cores are applications with data path linked into their address space.
- Worker flow:
  - Busy Poll -> Receive -> Run Request to Completion -> Transmit -> Repeat.
- Use a lean TCP implementation, zero copy.
- Queuing delay as a proxy for capacity crunch, allocate or shrink cores.
- Allocate cores every N us (respond quickly to load spikes).
- Work stealing.

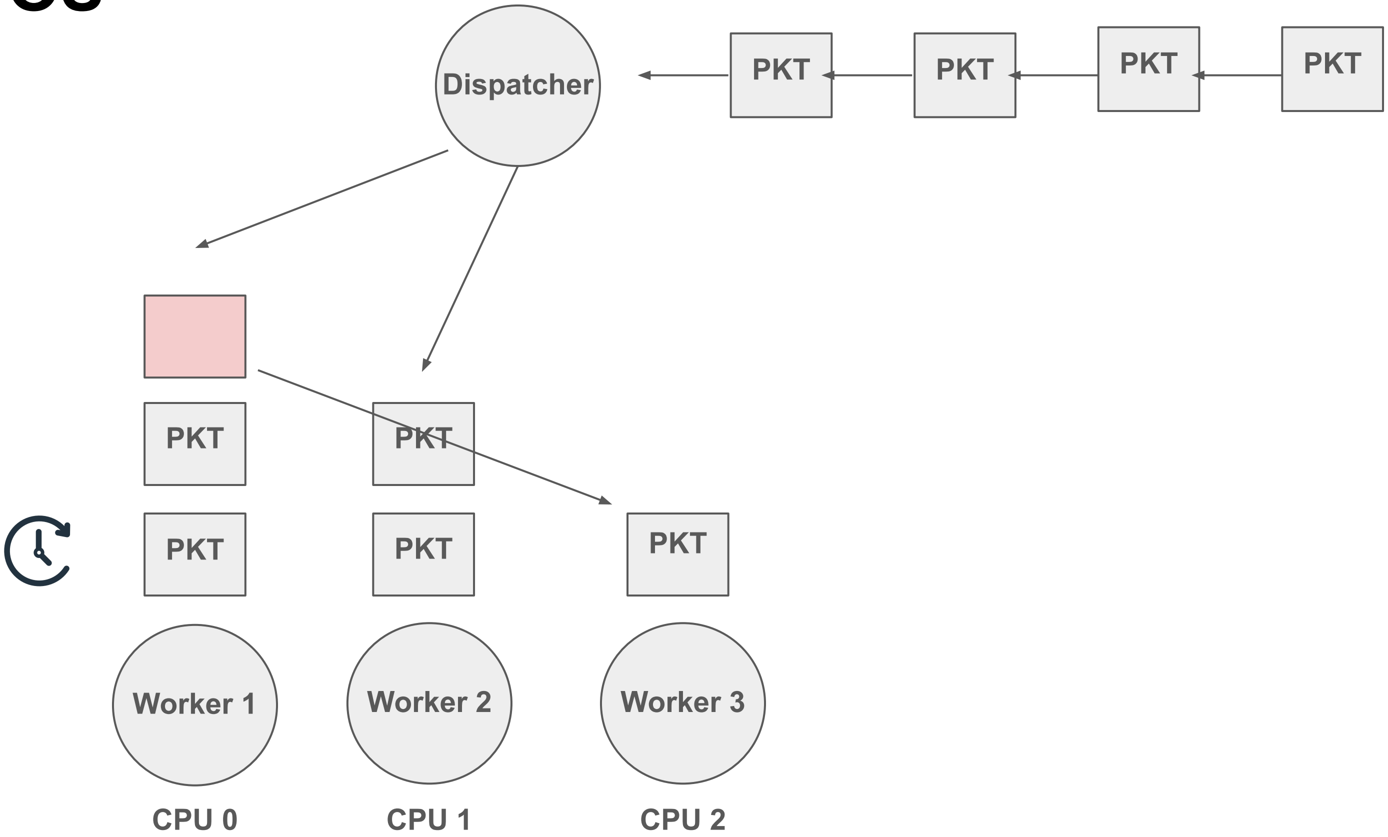
Dataplane OS



Dataplane OS



Dataplane OS



Role of Extensibility

- Reduces burden of experimentation.
- Faster feedback loop: rollout, testing, iteration.
- Adaptive.
- Deployable in production.

Role of Extensibility

- Reduces burden of experimentation.
- Faster feedback loop: rollout, testing, iteration.
- Adaptive.
- Deployable in production.

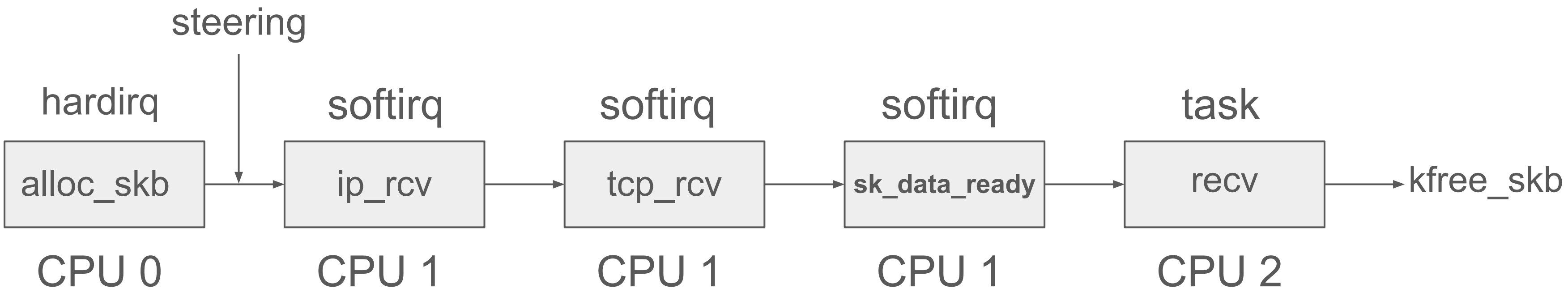
- Two paths to victory:
    - Experimentation motivates and leads to upstream changes.
    - Deliver benefits through extensions themselves.

Observations

- For each case, very little is changed about functionality.
  - E.g. networking processing logic mostly the same, or reused.
- Individual kernel processing steps remain the same.
- Scheduling of kernel work changes, or the execution context changes.
- Can we have a generic way of performing such modifications to the kernel?

Abstracting computation … safely

- Need a way to represent computation tied to a kernel object.
- Computation may happen at disparate locations in the kernel (in sequence).
- Resources acquired at individual steps may be released by later steps.
- Context and state of computation carried through each step.
- E.g. network Rx processing.

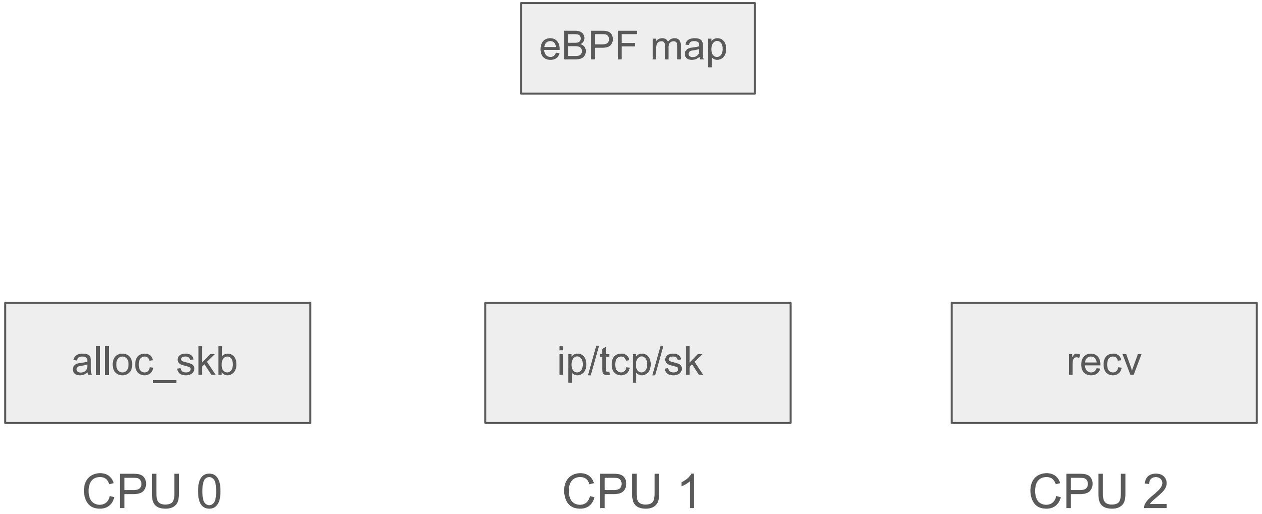


Fibers

- Sequential abstraction to represent possibly asynchronous work.
- Compiler converts a sequential function into a state machine.
  - Fibers are built on top of coroutines.
  - Mostly try to reuse LLVM’s support in BPF backend as far as implementation goes.

- Captures resources for the lifetime of processing in “fiber state”.
  - This is a well-known benefit, i.e. elimination of shared state.
  - C++ folks will recall shared_ptr<T> proliferation.
- Helps the verifier more than the user.

Yeah, maybe let’s not?



eBPF map

alloc_skb

ip/tcp/sk

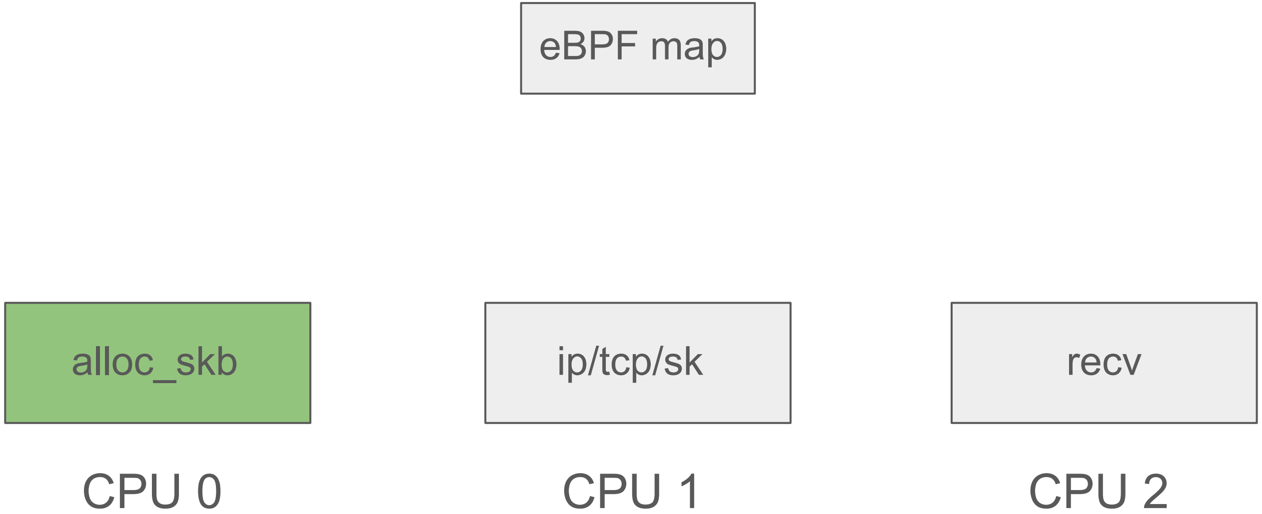
recv

CPU 0

CPU 1

CPU 2

Yeah, maybe let’s not?



eBPF map

alloc_skb

CPU 0

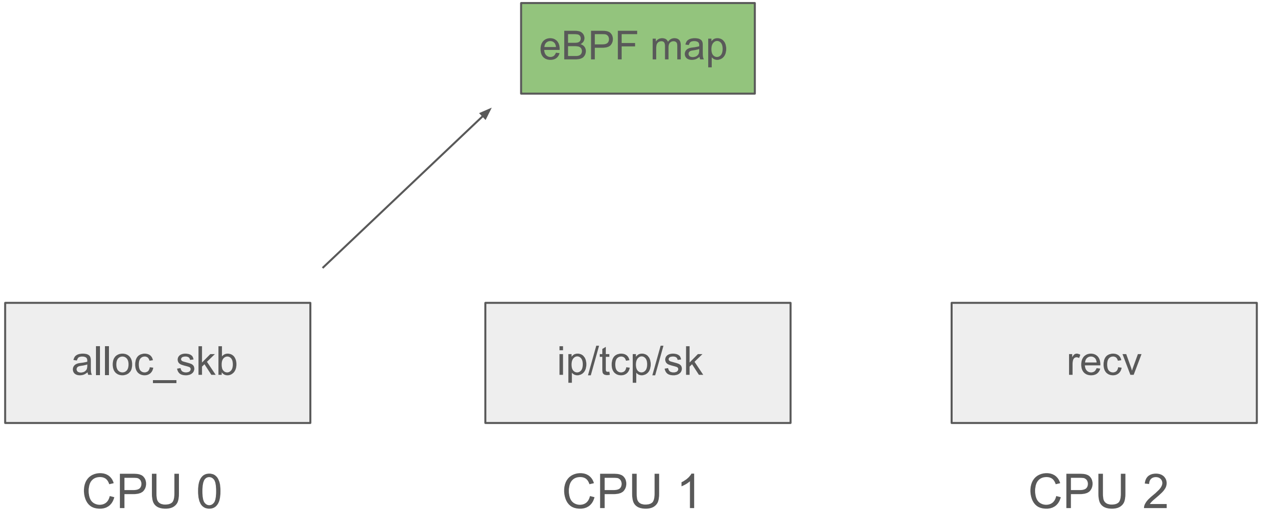
ip/tcp/sk

CPU 1

recv

CPU 2

Yeah, maybe let's not?


eBPF map

alloc_skb

ip/tcp/sk

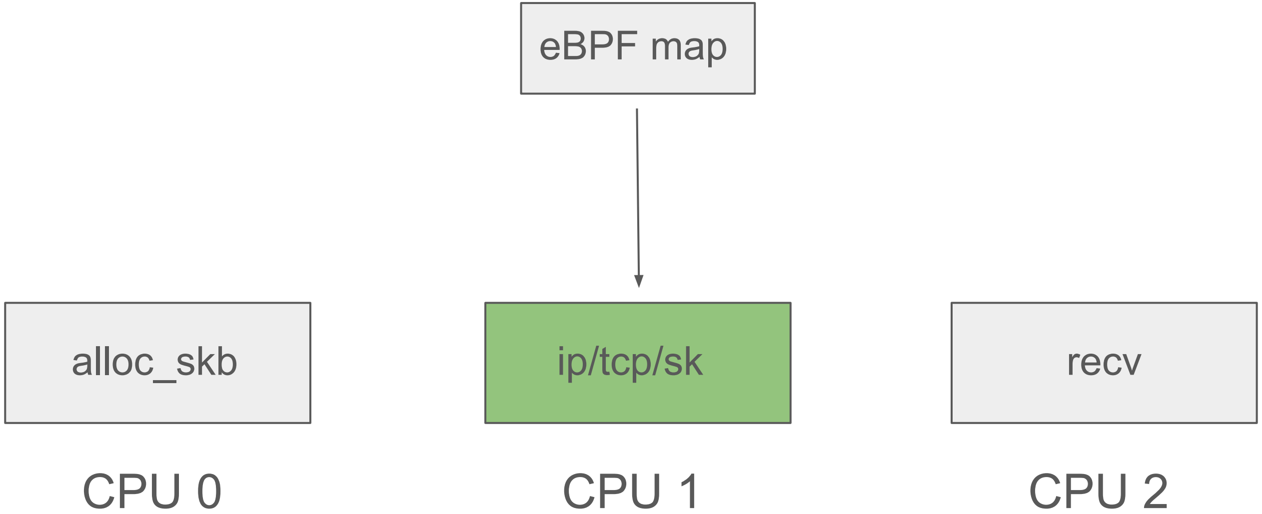
recv

CPU 0

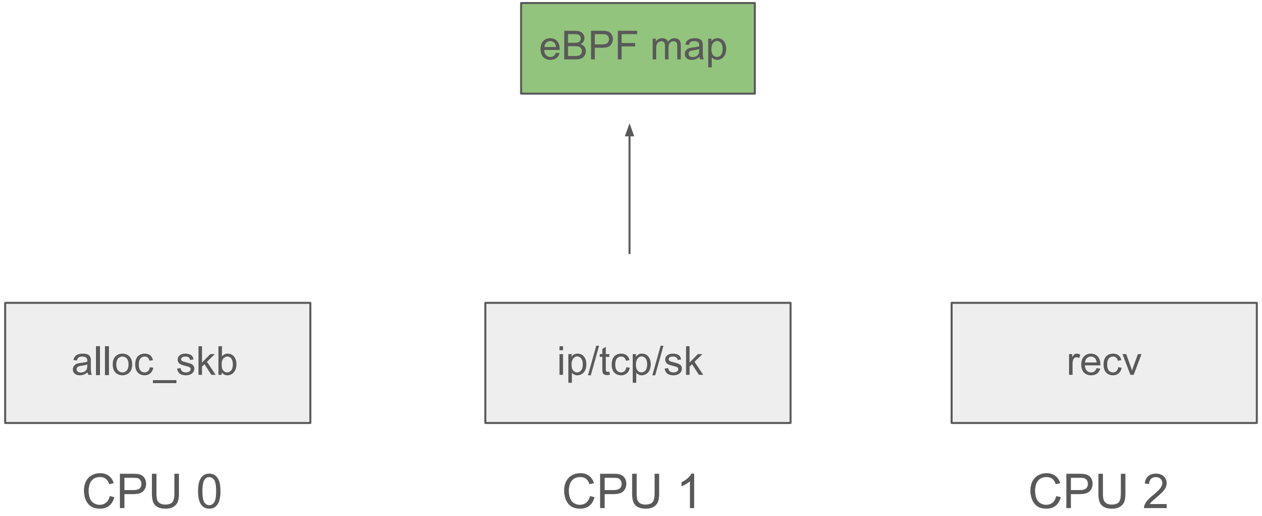
CPU 1

CPU 2

Yeah, maybe let’s not?



Yeah, maybe let’s not?



eBPF map

alloc_skb

ip/tcp/sk

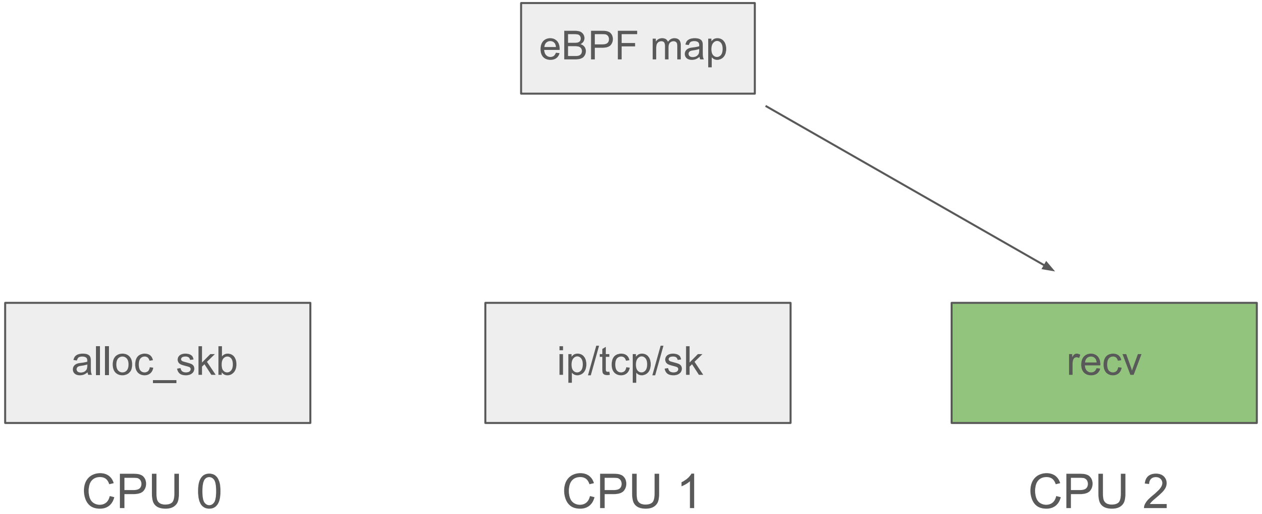
recv

CPU 0

CPU 1

CPU 2

Yeah, maybe let’s not?



eBPF map

alloc_skb

ip/tcp/sk

recv

CPU 0

CPU 1

CPU 2

Solution Sketch: TAPI

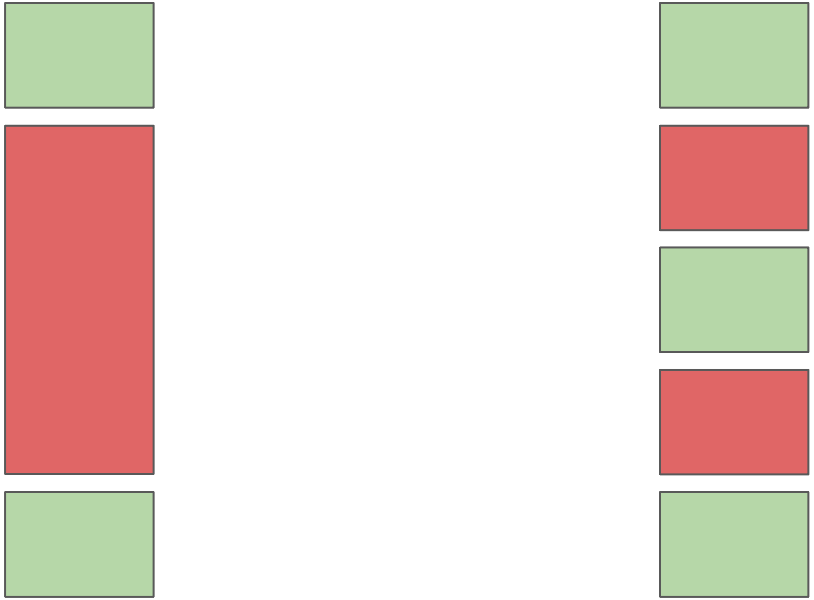
- Each frame corresponds to a fiber (since work done maps to each packet).
- TAPI reduced to implementing multiplexing of fibers on kernel threads.
- s/wq item/fiber/g
- s/workqueue/kthread pool/g

- BPF decides assignment of packet processing (i.e. fiber execution) to kernel threads.
- Work stealing to mitigate load imbalance.
- Collapse work onto same thread, spread to multiple kthreads.

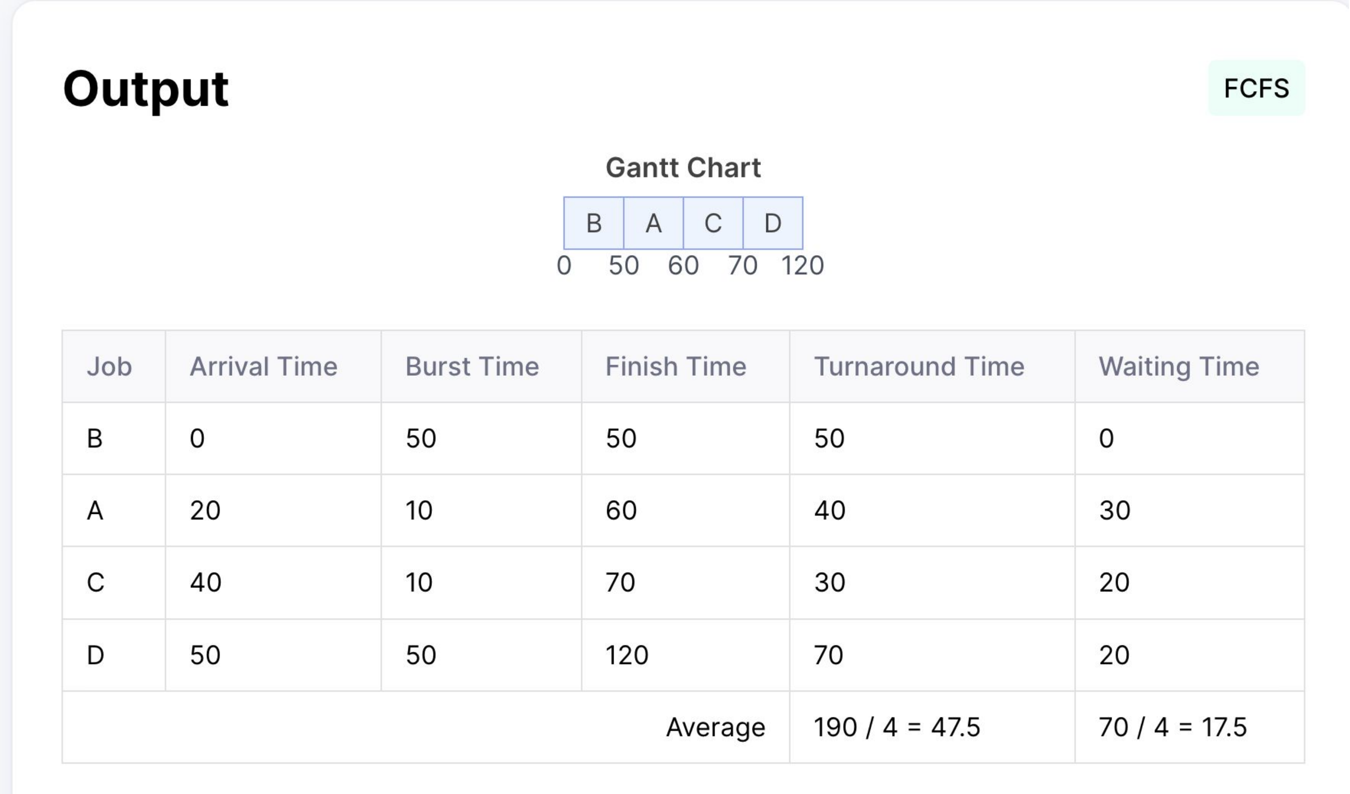
Solution Sketch: Isolation

- One possible solution: processor sharing at packet processing level.
- Fibers can allow interleaving processing.
- Network stack not ready for this yet, but should be possible.
- Preemption doesn’t have to be interrupt driven.
- Can be compiler driven approximation (yield points placed by approximating quantum). Works well with non-preemptive code.
  - Compiler Interrupts, Concord
- Coroutine switching in order of 10s of ns.

Processor Sharing



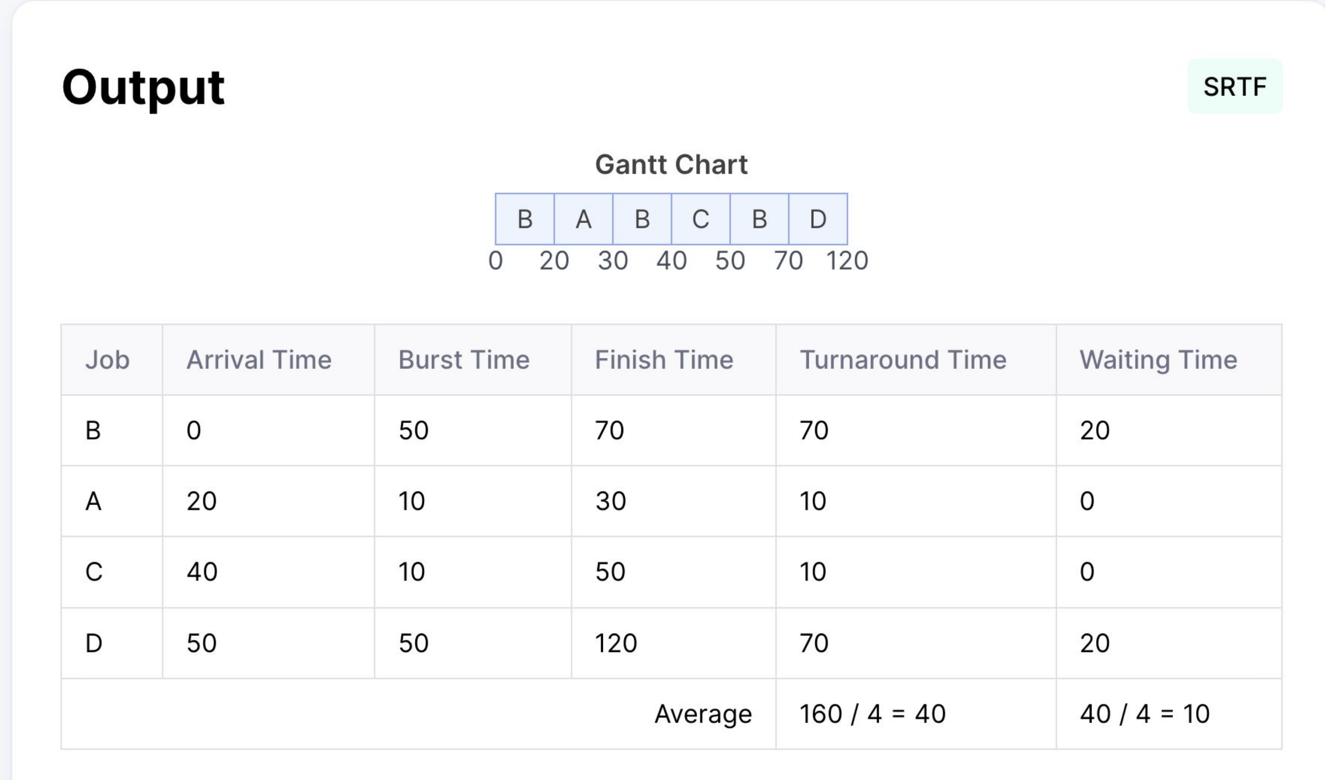
Napkin Math; FCFS; A, C - Short, L-Critical



Output

Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
B	0	50	50	50	0
A	20	10	60	40	30
C	40	10	70	30	20
D	50	50	120	70	20
			Average	190 / 4 = 47.5	70 / 4 = 17.5

SRTF (with Preemption)



Output

Job	Arrival Time	Burst Time	Finish Time	Turnaround Time	Waiting Time
B	0	50	70	70	20
A	20	10	30	10	0
C	40	10	50	10	0
D	50	50	120	70	20
			Average	160 / 4 = 40	40 / 4 = 10

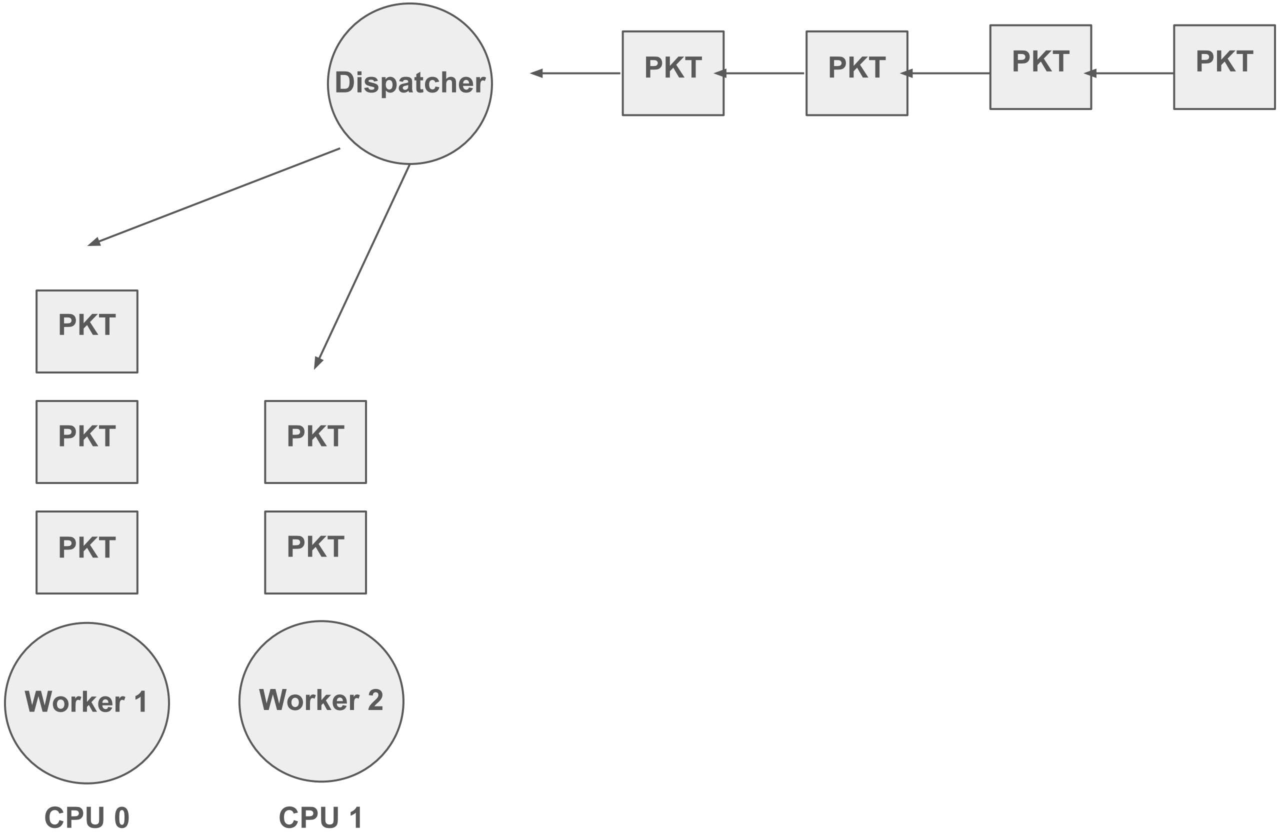
Solution Sketch: Dataplane OS

- Just like our example, construct a slim data path where for the data path:
    - Pages registered for ZC per NIC-queue, a. la. AF_XDP or ZC Rx.
    - Possibly do GRO (in BPF).
    - For TCP established state, carve out TCP sock state updates into a kfunc (both recv/send).
- Do not hit socket layer; do not build skb; only update `struct tcp_sock.`
- Yield to user space for application processing.
- Take over Tx processing.
- Go through qdisc layer (e.g. we still might want bw management).

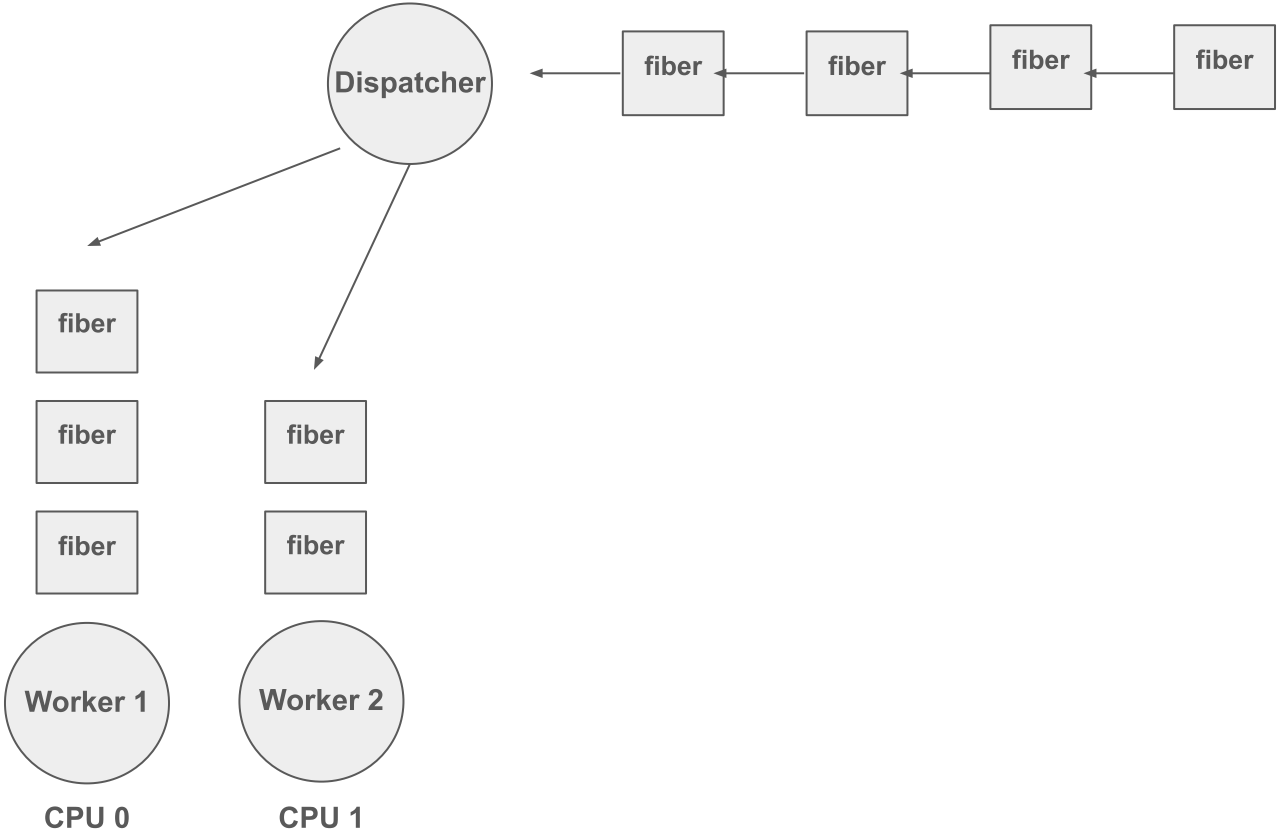
Solution Sketch: Dataplane OS

```
int data_path(struct xdp_md *ctx) {
    if (bpf_sk_lookup(...)->state != TCP_ESTABLISHED)
        return XDP_PASS;
    if (bpf_gro_xdp(ctx) // Custom BPF GRO engine
        co_return 0;
    tcp_rcv_established(ctx);   // Returns before sk_data_ready
    co_await yield_to_user();   ←———————————  Suspension point
    tcp_send(ctx);                         // Update tcp_sock
    co_return enqueue_qdisc(ctx);// We still want host-wide bw management
}
```

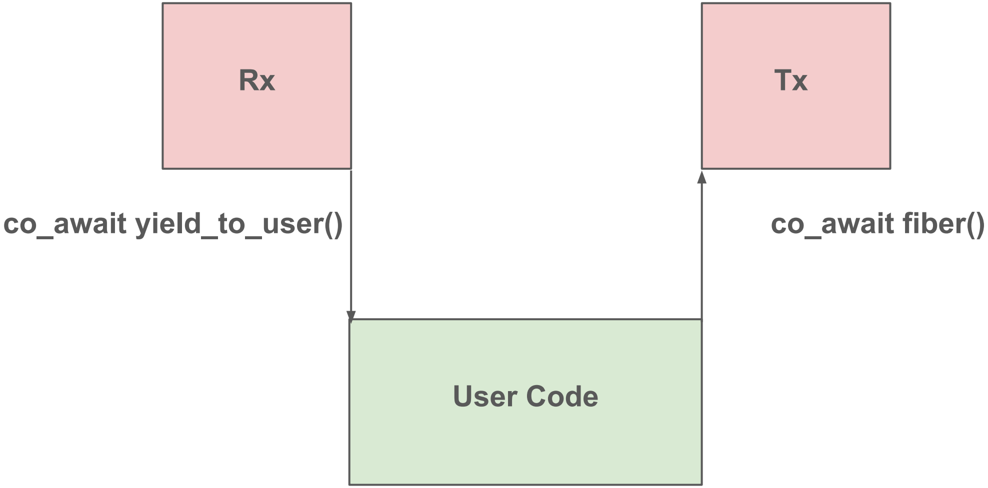
Dataplane OS



Dataplane OS



Cross Section



Rx

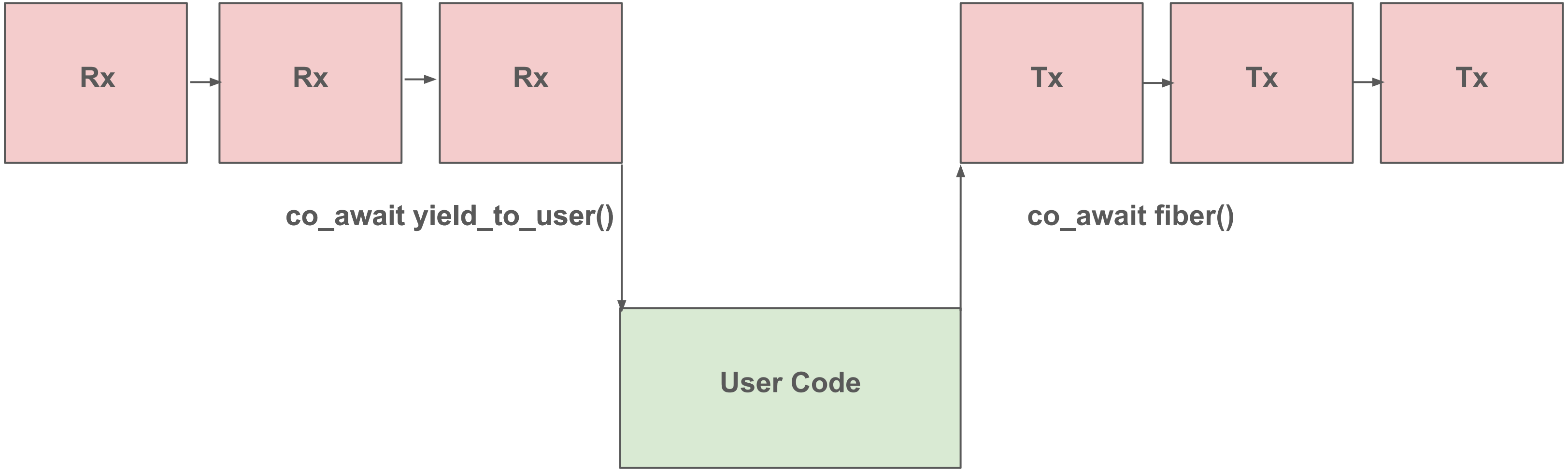
Tx

co_await yield_to_user()

co_await fiber()

User Code

Cross Section - Batching



Rx → Rx → Rx

co_await yield_to_user()

User Code

co_await fiber()

Tx → Tx → Tx

Solution Sketch: Dataplane OS

- What happens if I free the fiber after yield to user space?
  - XDP frame is part of fiber’s state after suspension.
  - So, would be released with the fiber’s destruction.
- What happens if I don’t run the fibers to completion?
  - You have a queue build up, the connection becomes non-responsive.
  - Same as what happens when a server stops reading from its socket.

Necessary, but not sufficient.

- We build upon the kind of functional extensibility eBPF supports.
    - XDP
    - TC
    - sched-ext
    - …

- Changing the structure / form of a subsystem goes hand in hand with functional extensibility.

Key Takeaways

- Kernel’s logic reused or repurposed.
  - TAPI, Isolation: Networking stack traversal remains the same.
  - Dataplane OS: Exclude unneeded stuff, keep the rest the same.

Key Takeaways

- Kernel’s logic reused or repurposed.
  - TAPI, Isolation: Networking stack traversal remains the same.
  - Dataplane OS: Exclude unneeded stuff, keep the rest the same.

- Fibers abstract kernel’s computation across execution contexts.
  - Use wholesale (TAPI), or pick and choose individual steps (Dataplane OS).
  - Resources acquired or used during execution tied to the fiber’s state.
  - Verifier gains visibility into end-to-end processing, reasons about safety.

Key Takeaways

- Kernel’s logic reused or repurposed.
    - TAPI, Isolation: Networking stack traversal remains the same.
    - Dataplane OS: Exclude unneeded stuff, keep the rest the same.

- Fibers abstract kernel’s computation across execution contexts.
    - Use wholesale (TAPI), or pick and choose individual steps (Dataplane OS).
    - Resources acquired or used during execution tied to the fiber’s state.
    - Verifier gains visibility into end-to-end processing, reasons about safety.

- Changes how things are scheduled, executed, preempted.
    - Computation when data flows through the system abstracted as an entity: fibers.
    - Driven to completion by a real schedulable entity: threads.
    - Can be preempted, perform symmetric transfer to other fibers, etc.

Questions?