PeeR: First-Class Scheduling for Latency-Critical eBPF Applications

Jeremy Carin Ben Holmes Weiyang “Frank” Wang

Ankit Bhardwaj Manya Ghobadi

MIT CSAIL Computer Science & Artificial Intelligence Laboratory

Tufts University

Systalyze



eBPF has gotten popular!

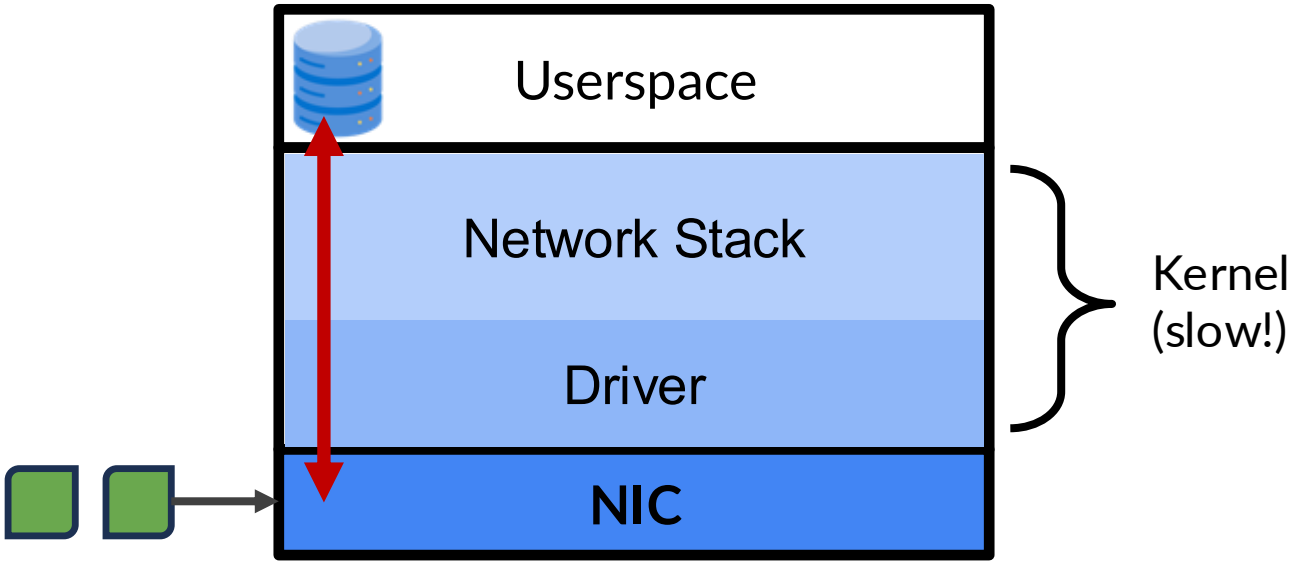
Observability

Extensibility

Performance

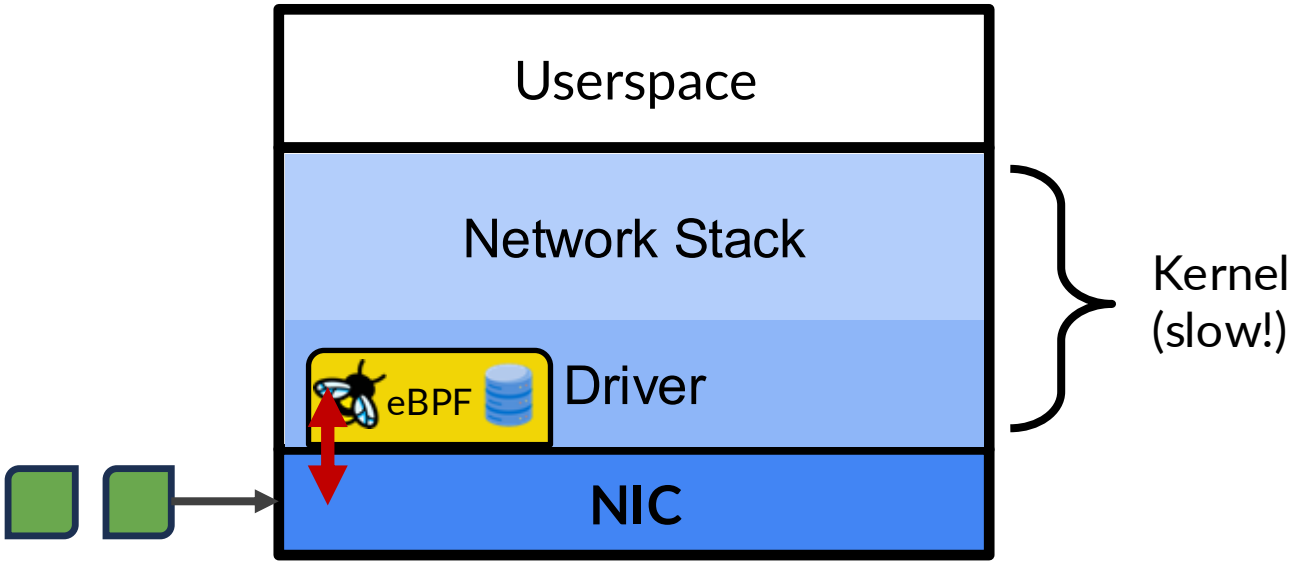
The promise of eBPF: performance with colocation

eBPF allows applications to offload logic into the kernel



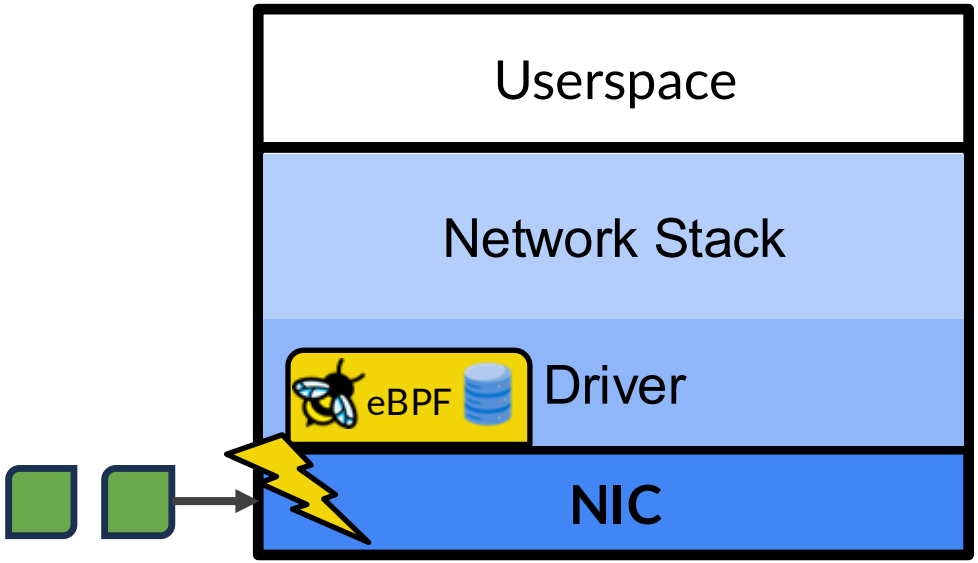
The promise of eBPF: performance with colocation

eBPF allows applications to offload logic into the kernel



The promise of eBPF: performance with colocation

- I/O based eBPF applications are **interrupt-driven**
- Runs in **interrupt context**
- **Run-to-completion** execution model

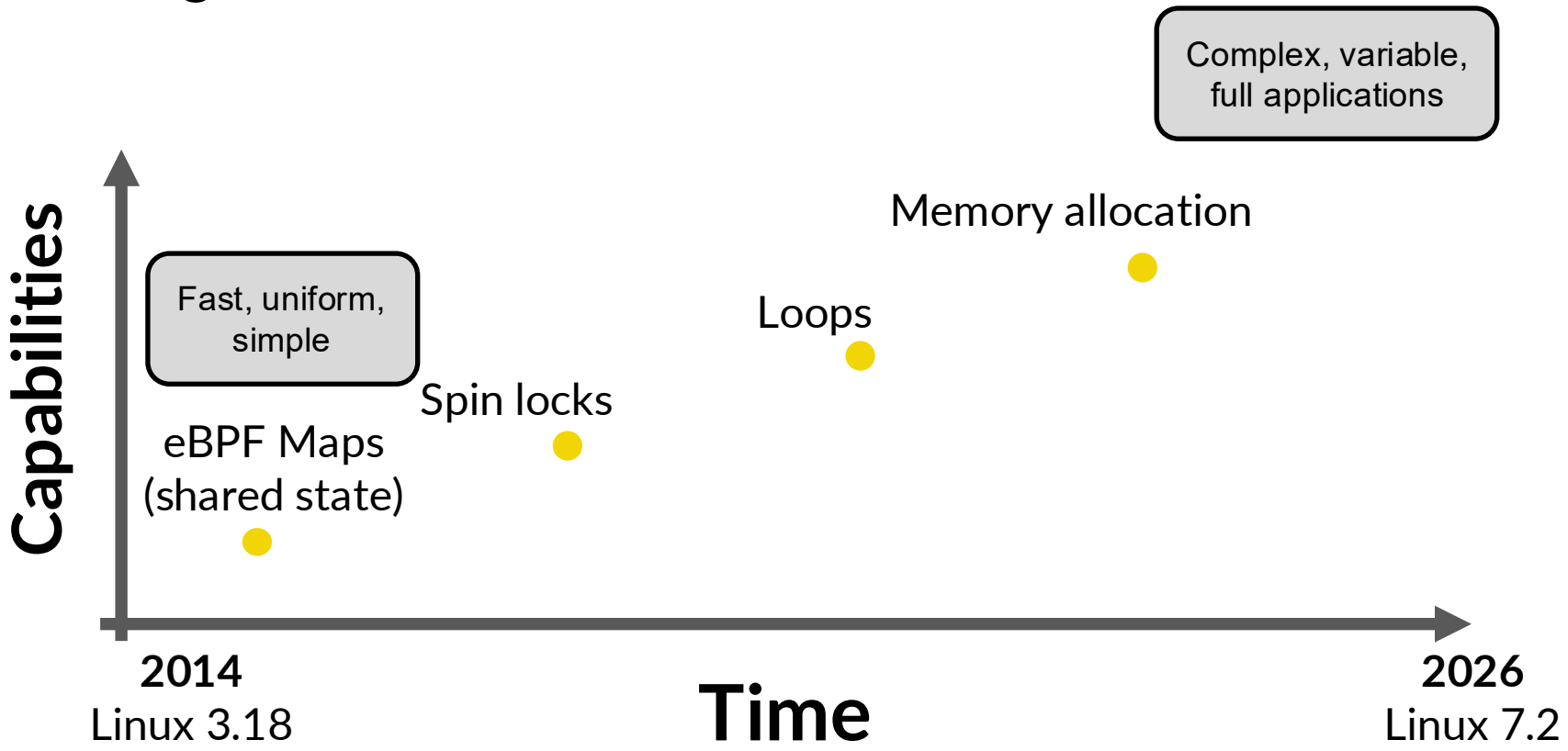


As a result of an interrupt-driven runtime, eBPF should provide **performance** and **colocation**.

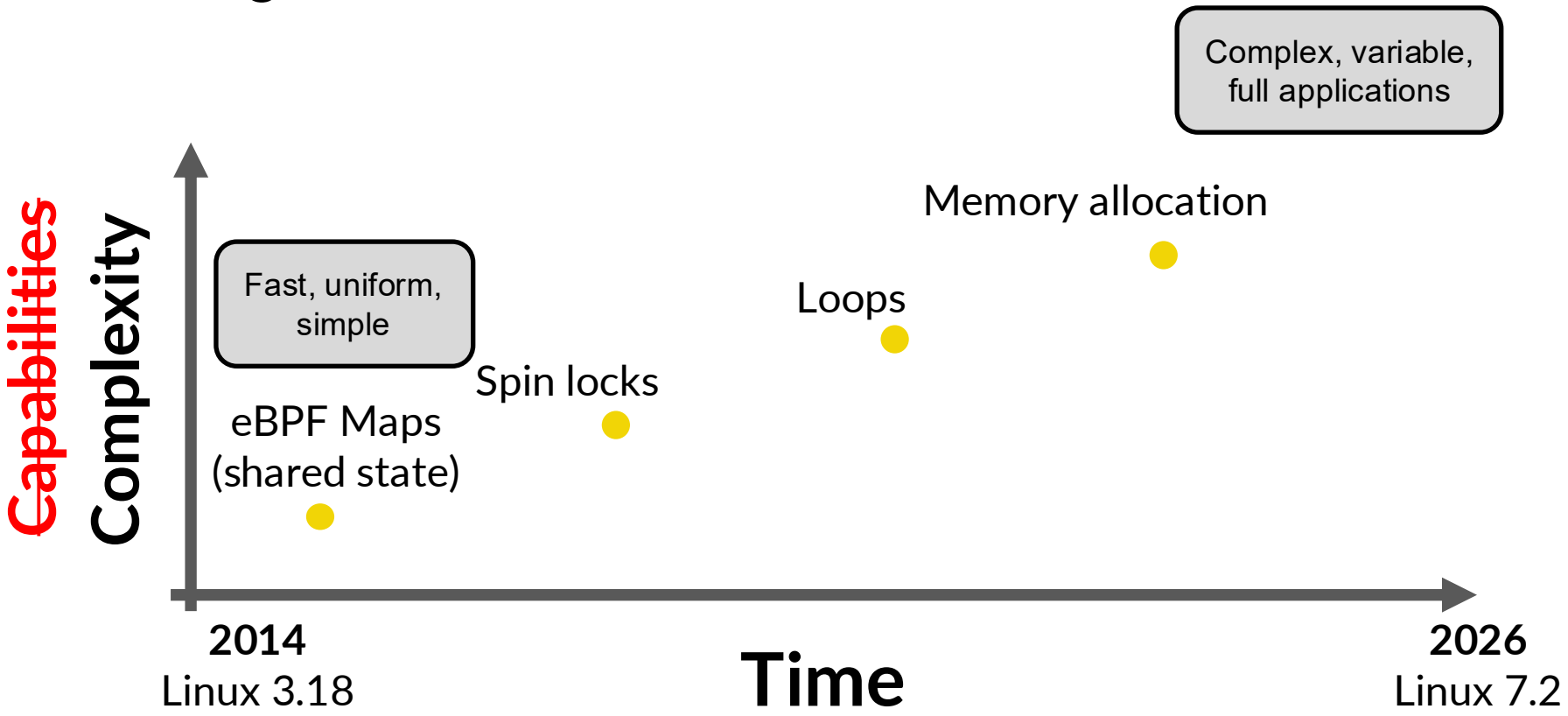
Performance: interrupts are fast

Colocation: event-driven

eBPF's growth

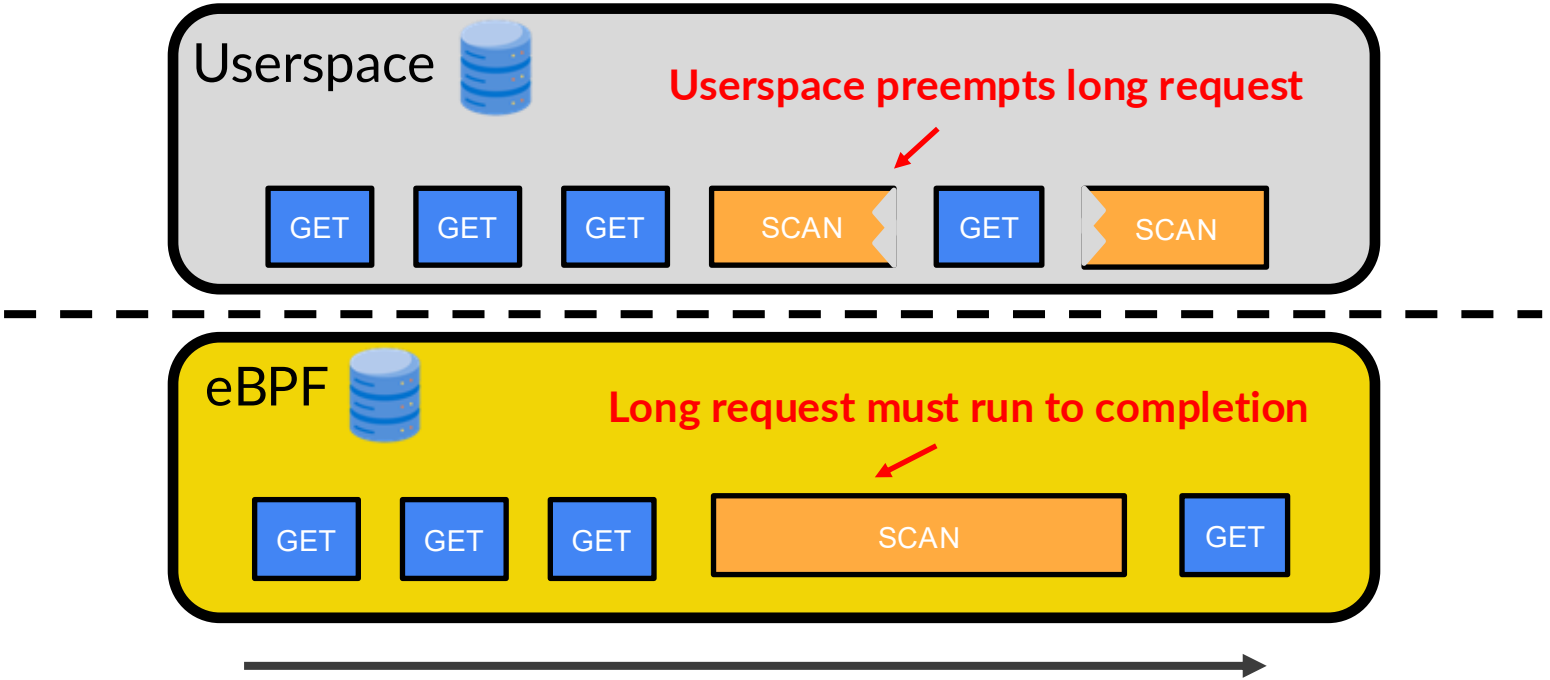


eBPF's growth

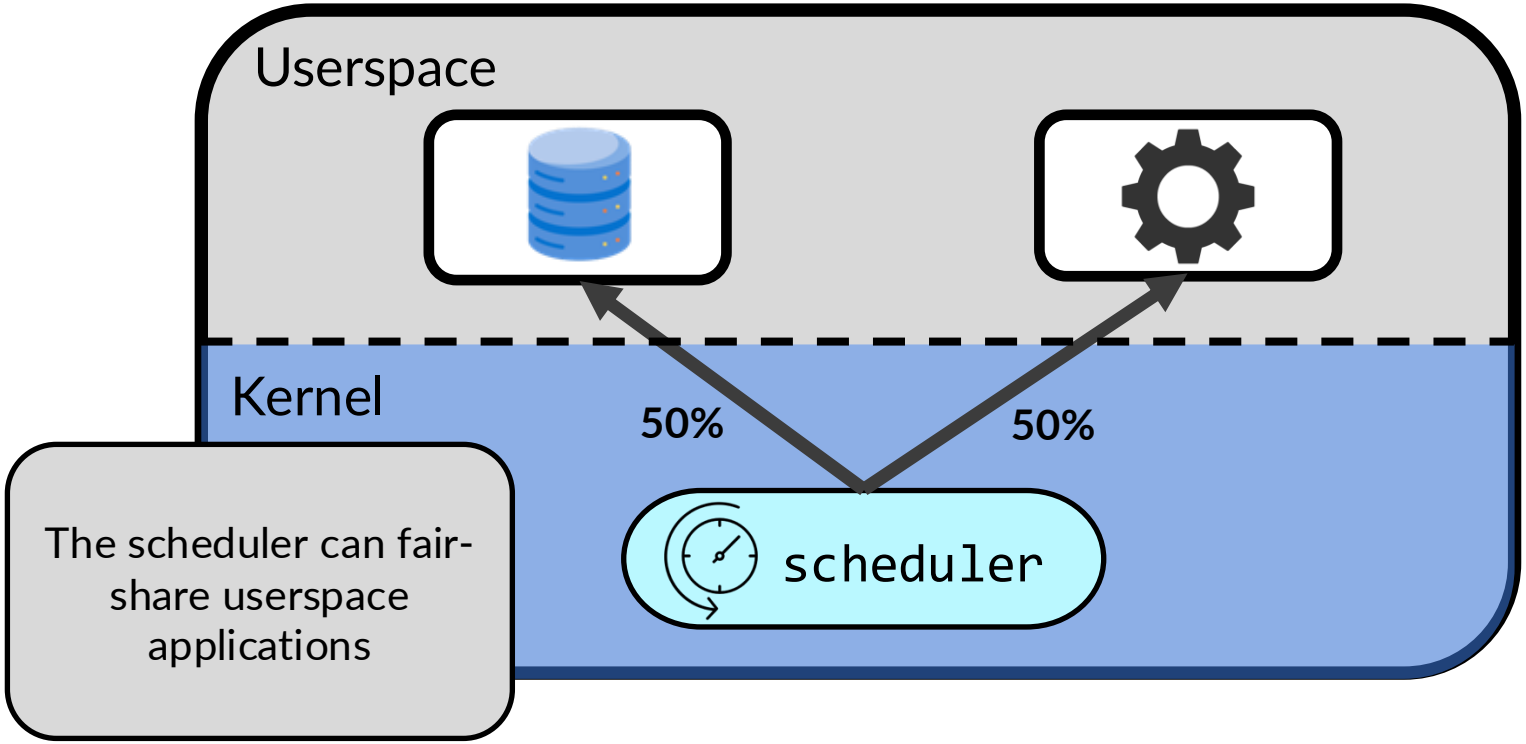


Problem: **intra**-application contention

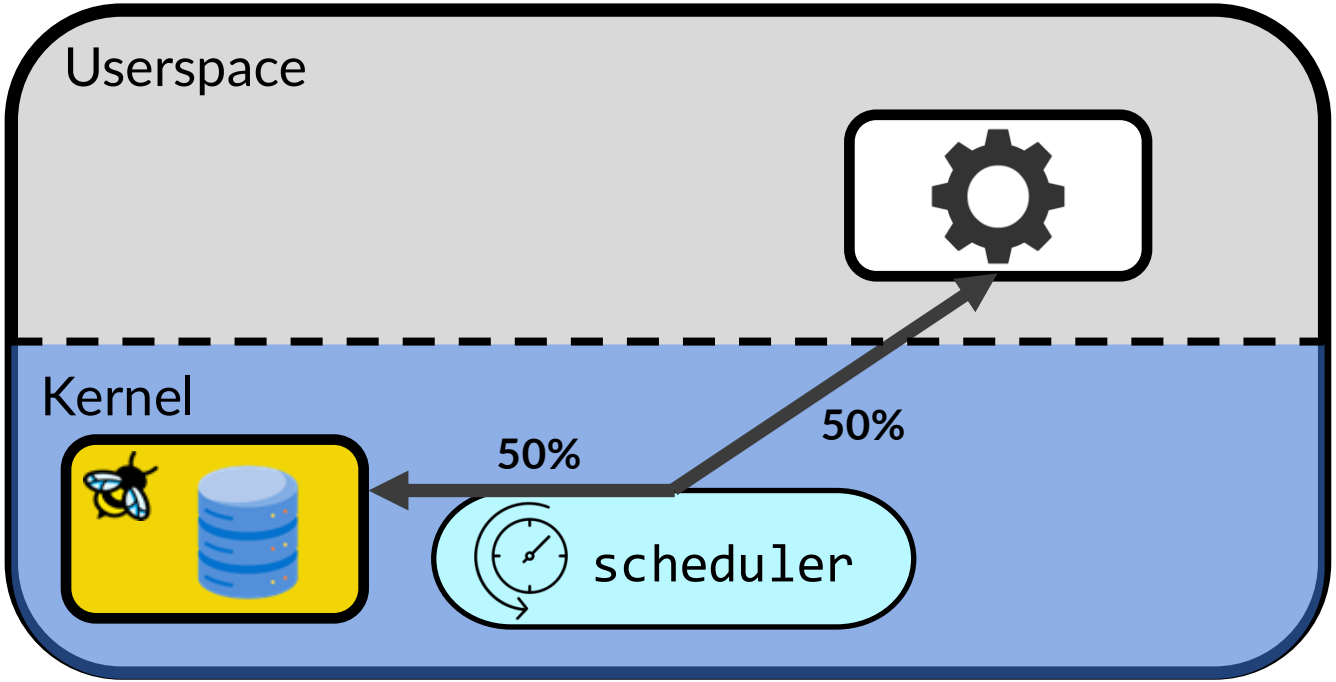
New capabilities allow for non-uniform request distributions; creates head of line blocking



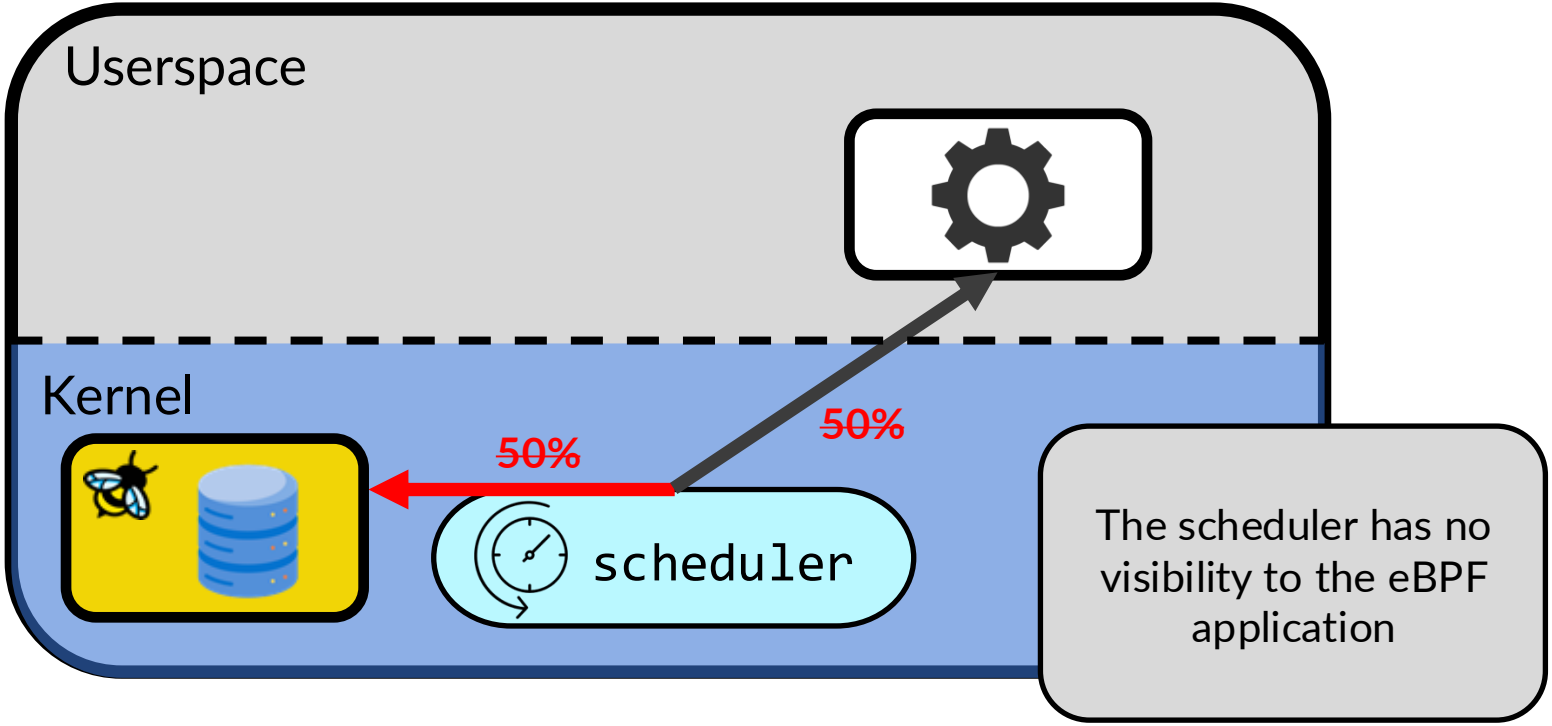
Problem: **inter**-application contention



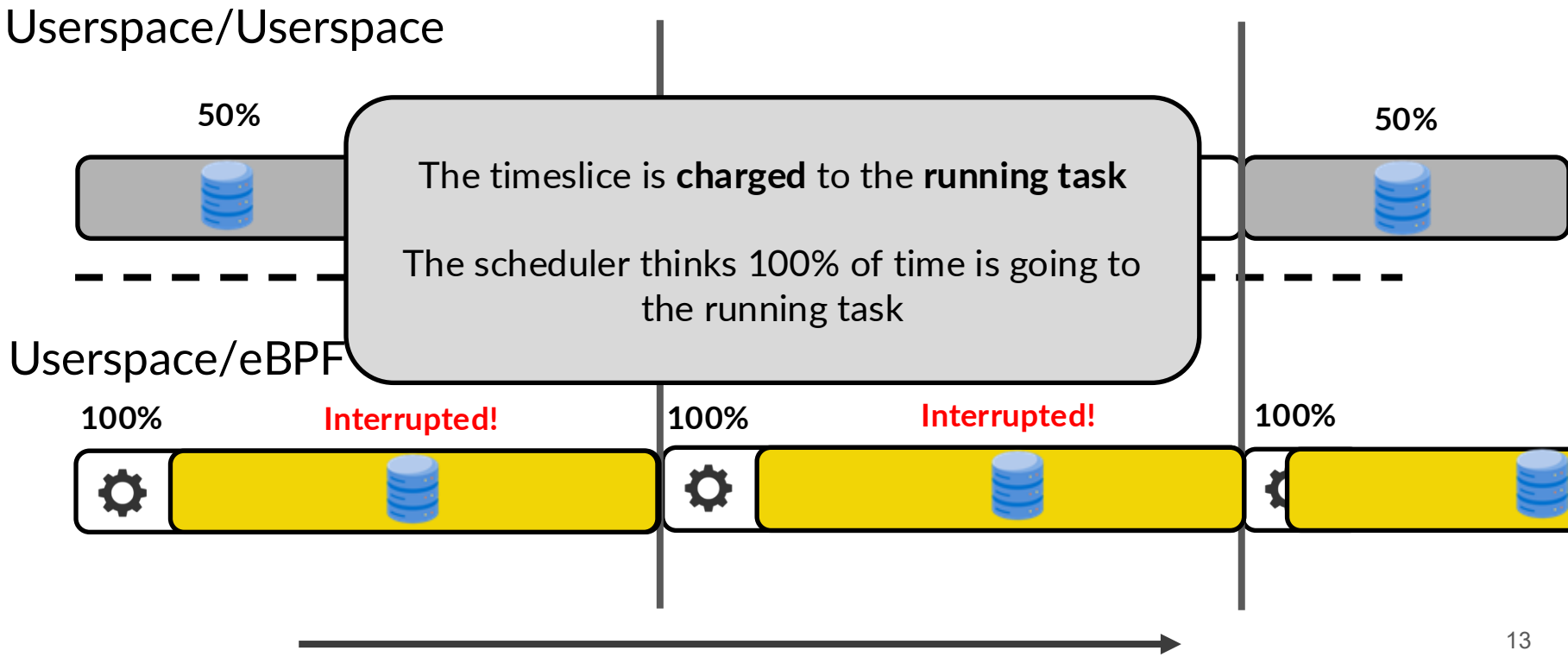
Problem: **inter**-application contention



Problem: **inter**-application contention



Problem: **inter**-application contention



eBPF should provide **performance** and **colocation**, but its run-to-completion model prevents both.

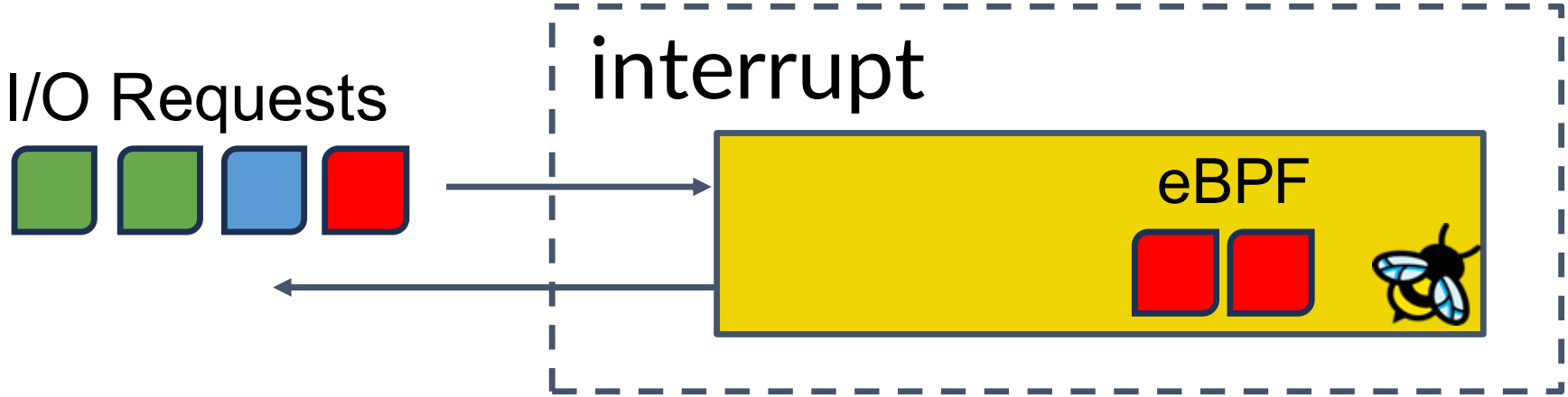
PeeR: Preemptable **e**BPF **e**xecution **R**untime
bringing schedulability to eBPF

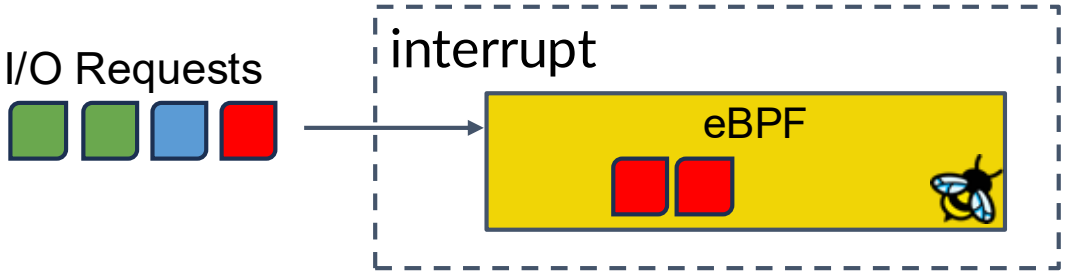
Performance: eliminate head-of-line blocking through preemption

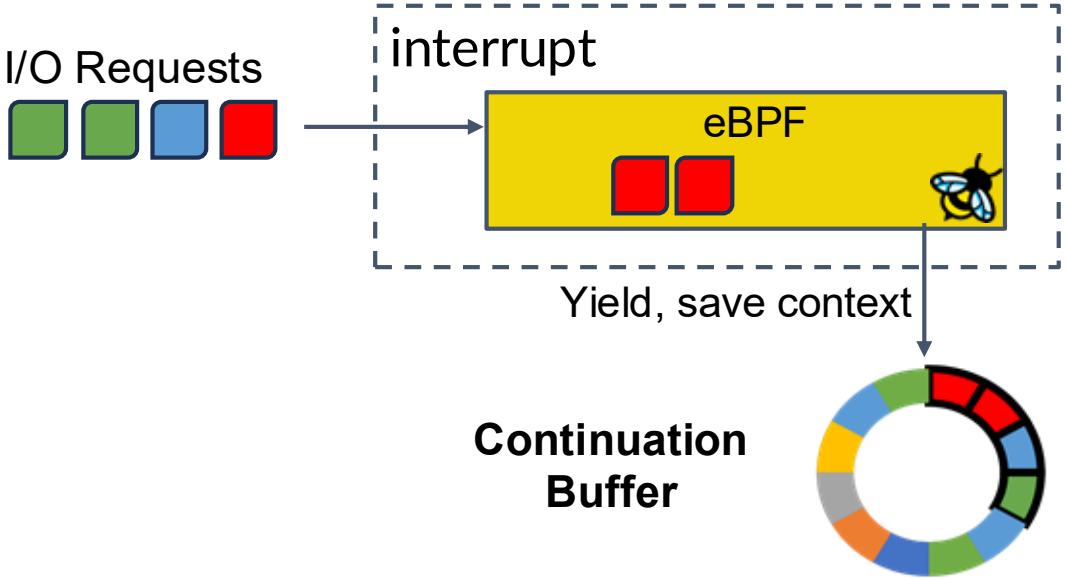
Colocation: expand domain of scheduler to eBPF applications

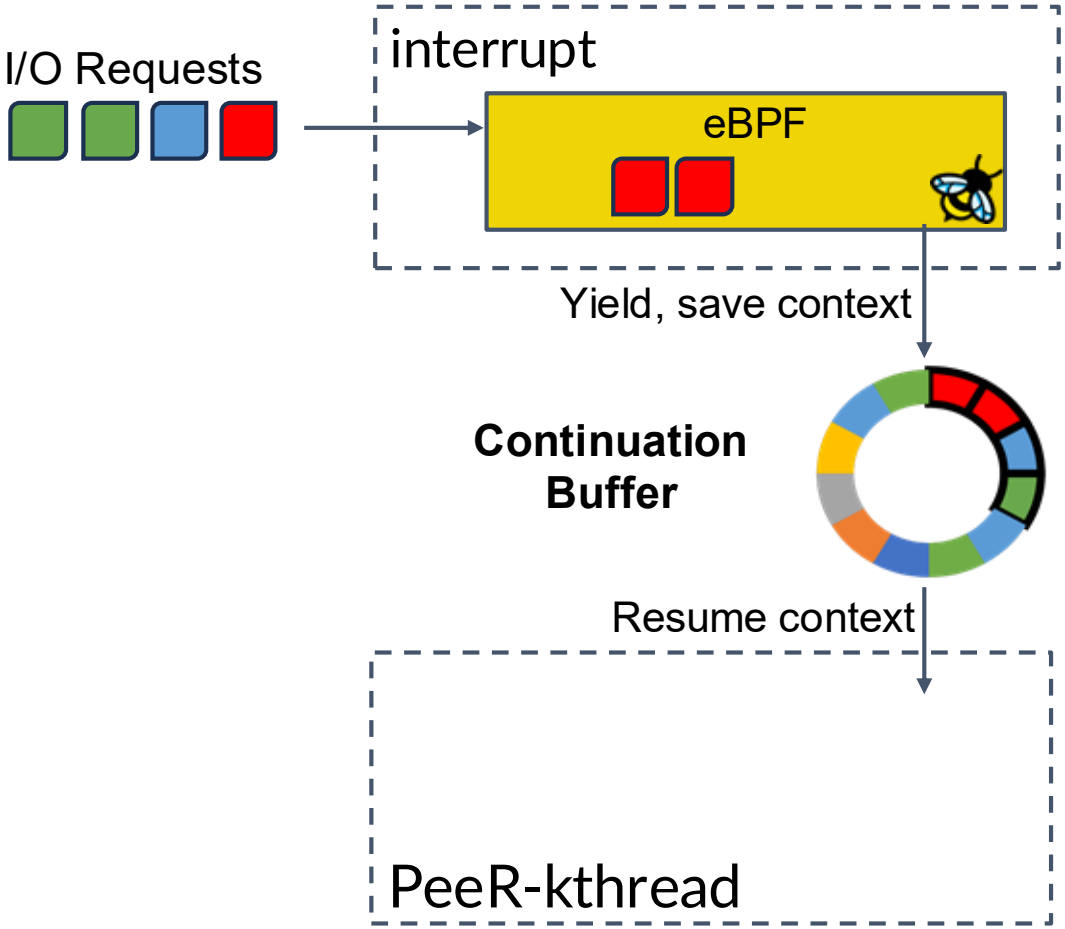
PeeR Design

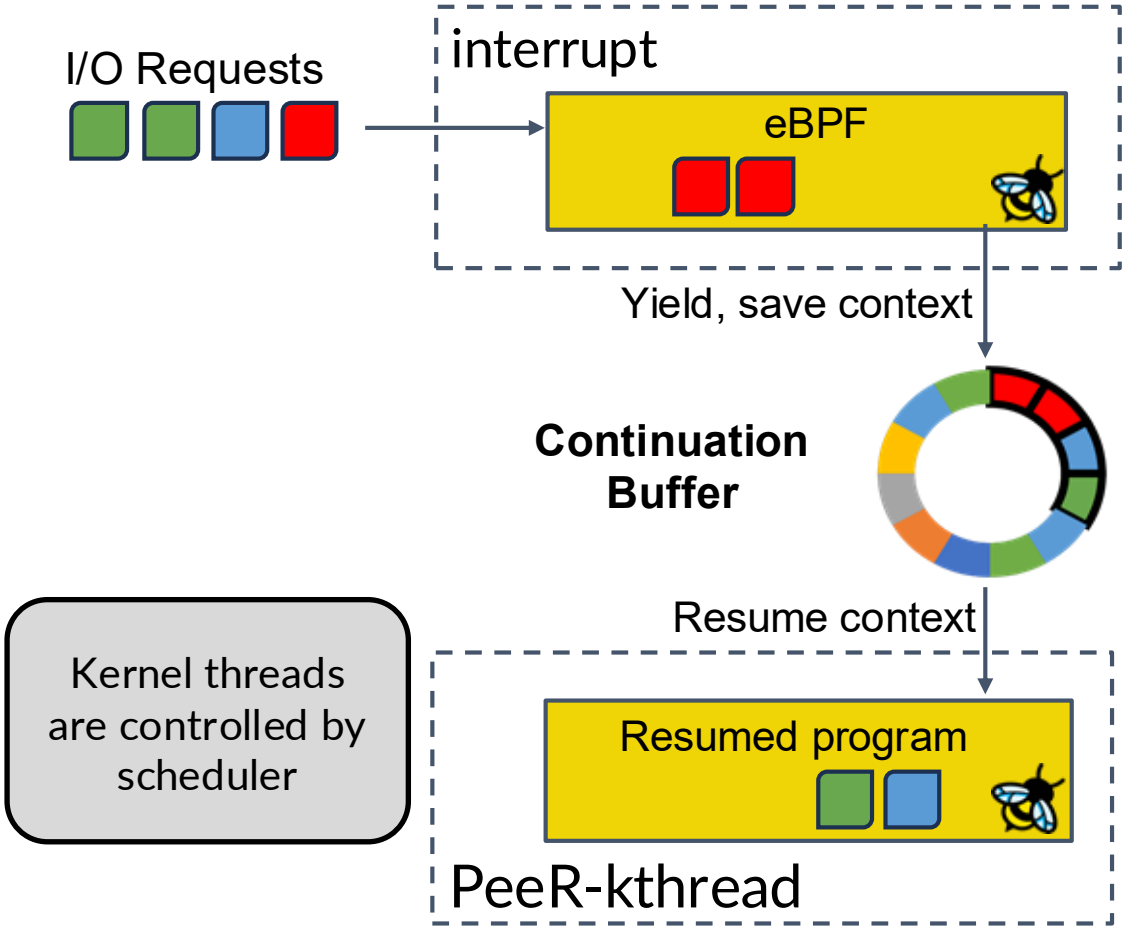
Normal eBPF path

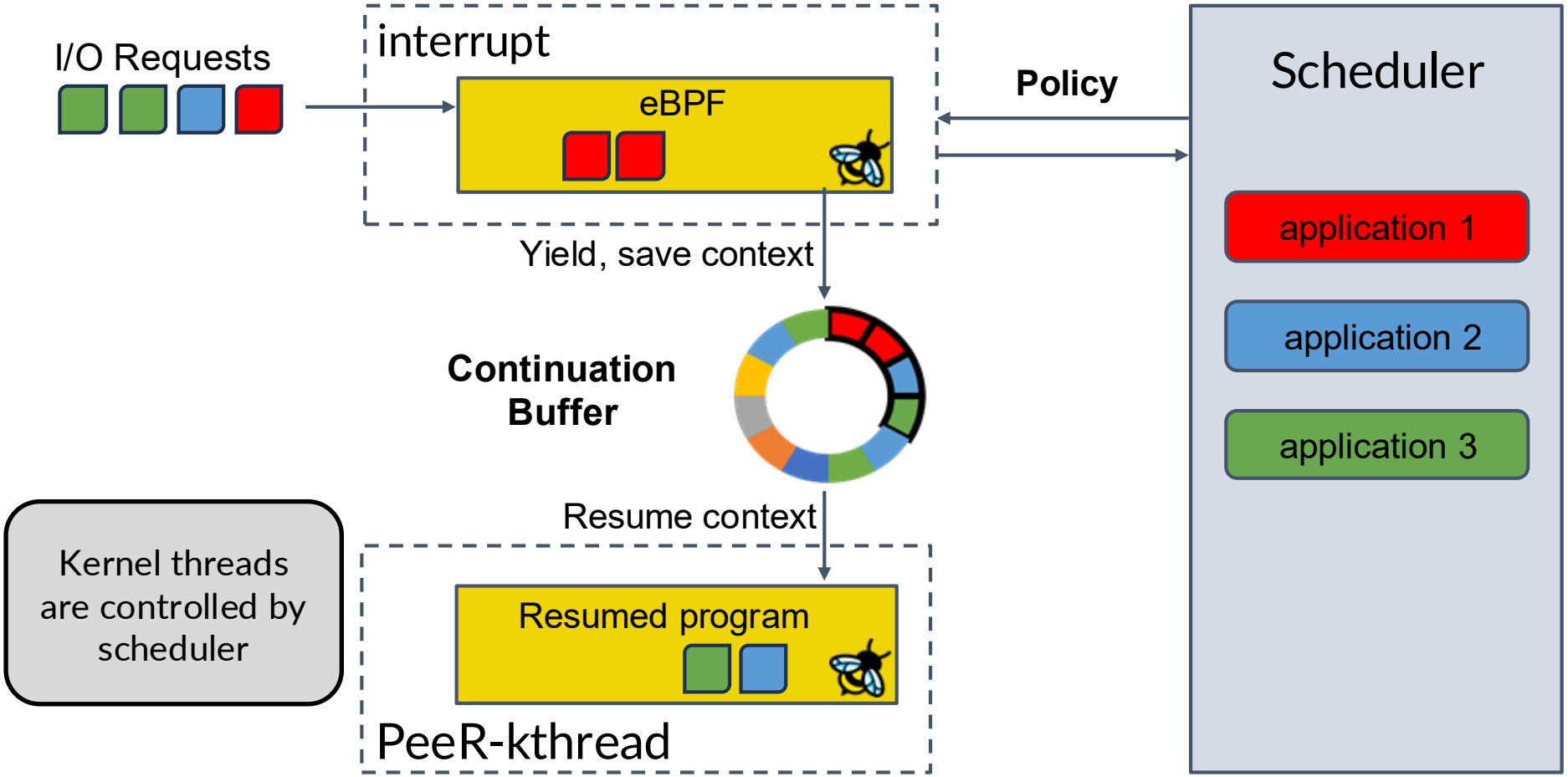


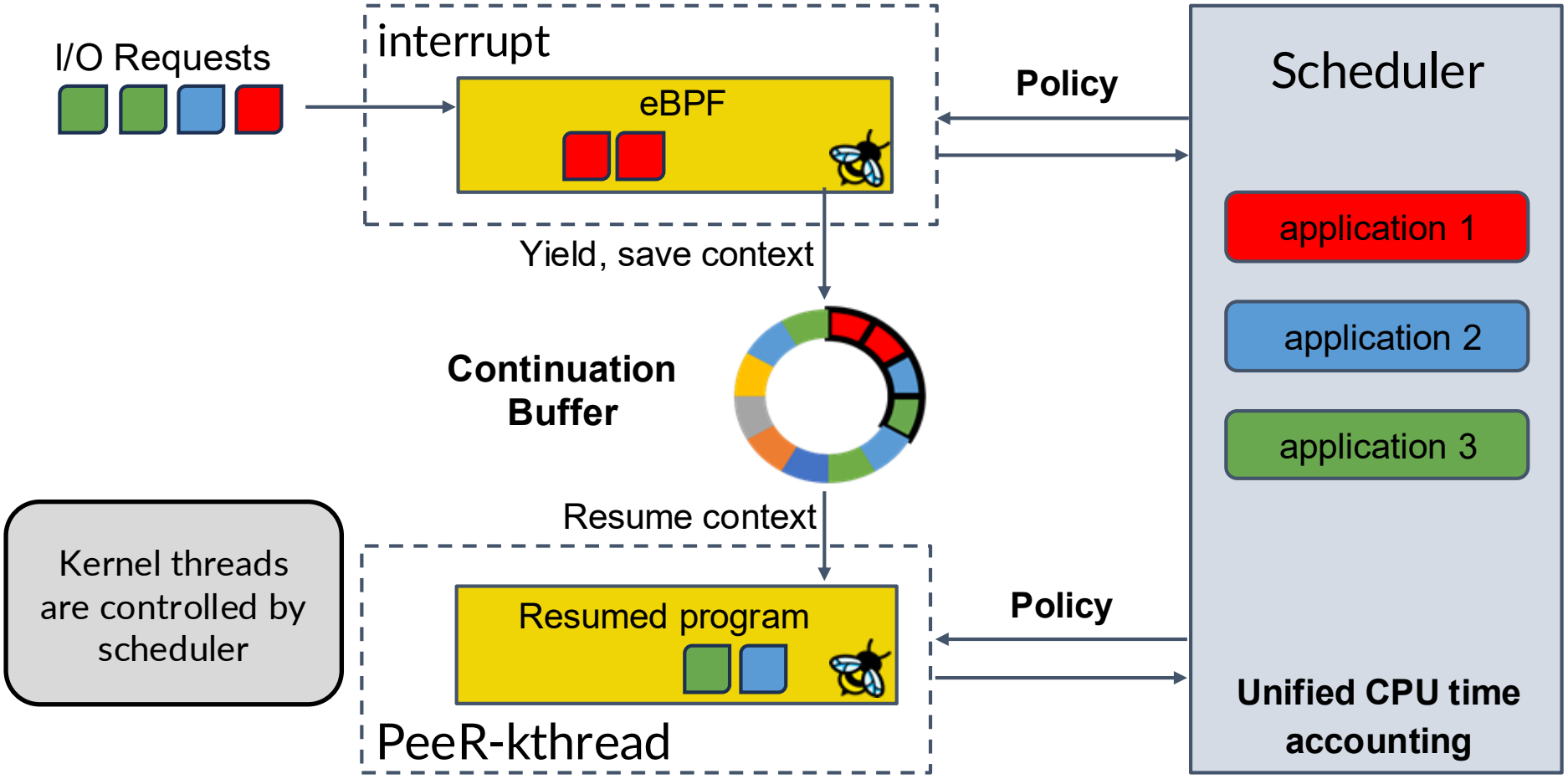

I/O Requests
interrupt
eBPF


I/O Requests
interrupt
eBPF
Yield, save context
Continuation Buffer


I/O Requests
interrupt
eBPF
Yield, save context
Continuation Buffer
Resume context
PeeR-kthread


I/O Requests
interrupt
eBPF
Yield, save context
Continuation Buffer
Resume context
Resumed program
PeeR-kthread
Kernel threads are controlled by scheduler

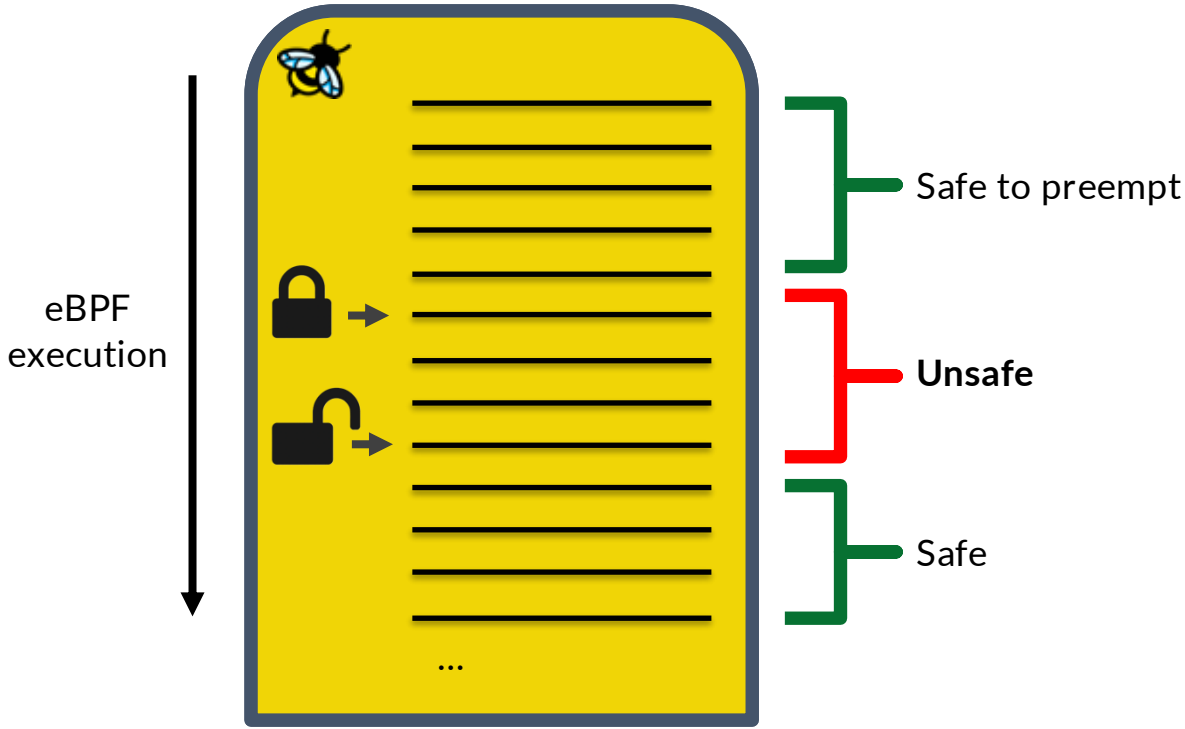

I/O Requests
interrupt
eBPF
Policy
Scheduler
application 1
application 2
application 3
Yield, save context
Continuation Buffer
Resume context
Resumed program
PeeR-kthread
Kernel threads are controlled by scheduler


I/O Requests
interrupt
eBPF
Policy
Scheduler
application 1
application 2
application 3
Yield, save context
Continuation Buffer
Resume context
Kernel threads are controlled by scheduler
Resumed program
Policy
PeeR-kthread
Unified CPU time accounting

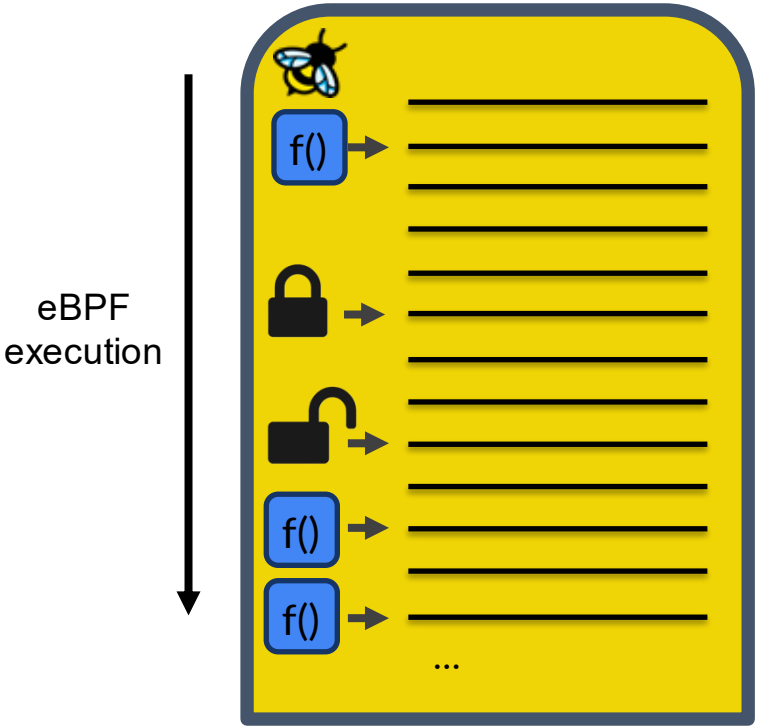
PeeR overview

- Keep eBPF’s interrupt path for fast requests, preempt long running invocations
- Resume in scheduler-controlled kernel thread context
- Preemption and resumption managed by scheduler

Challenge 1: Safe preemption



Solution: Utilize safety properties of helper functions

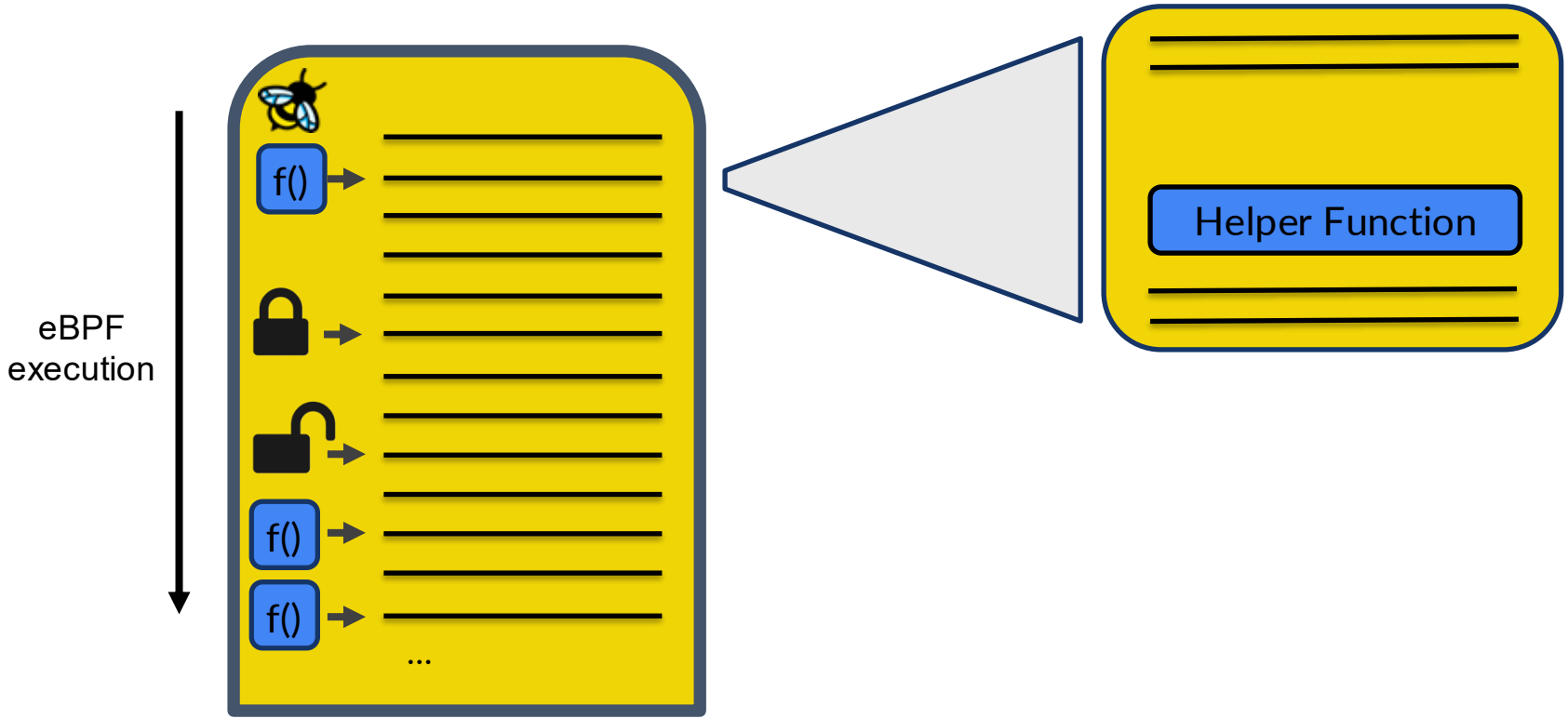


eBPF programs must use **helper functions** to interact with the kernel

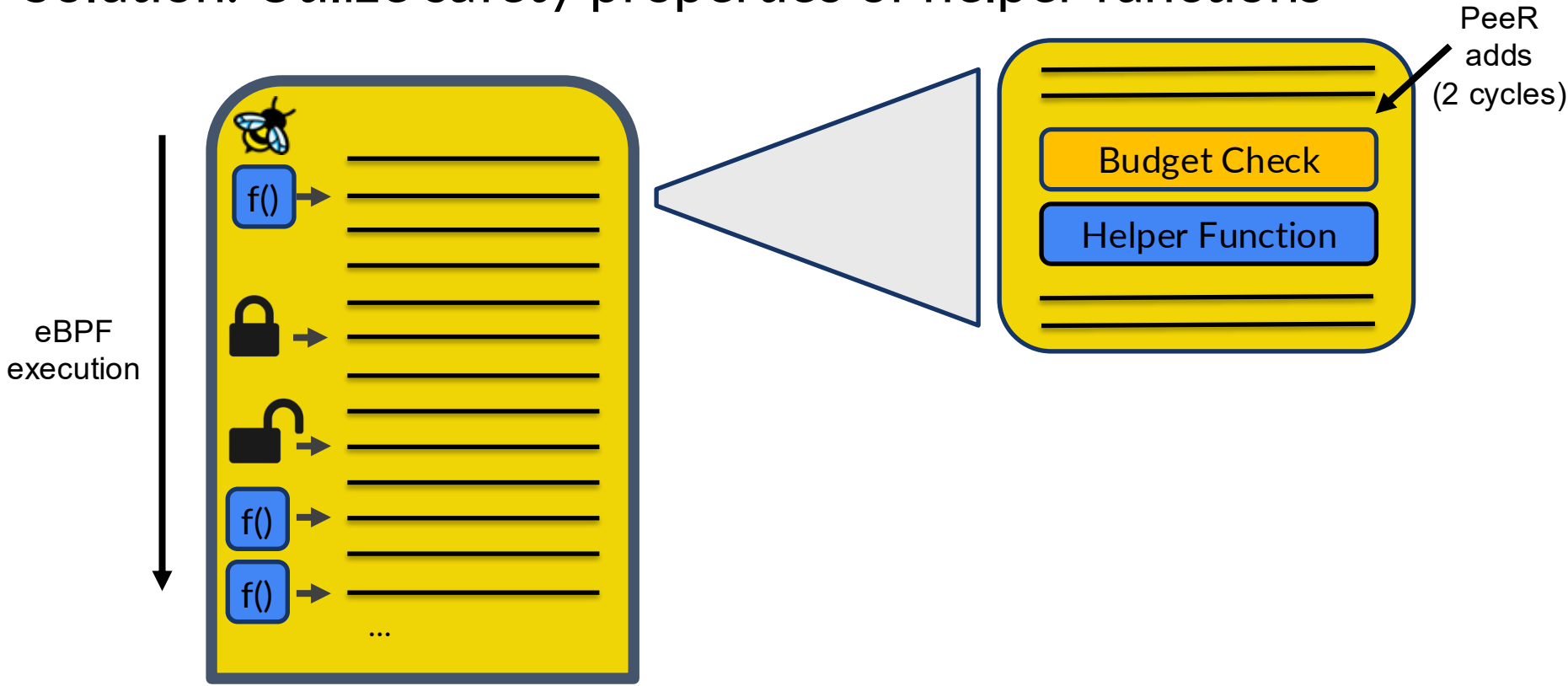
Helper functions are **inherently safe** points during execution

(cannot be used within locks, known register state)

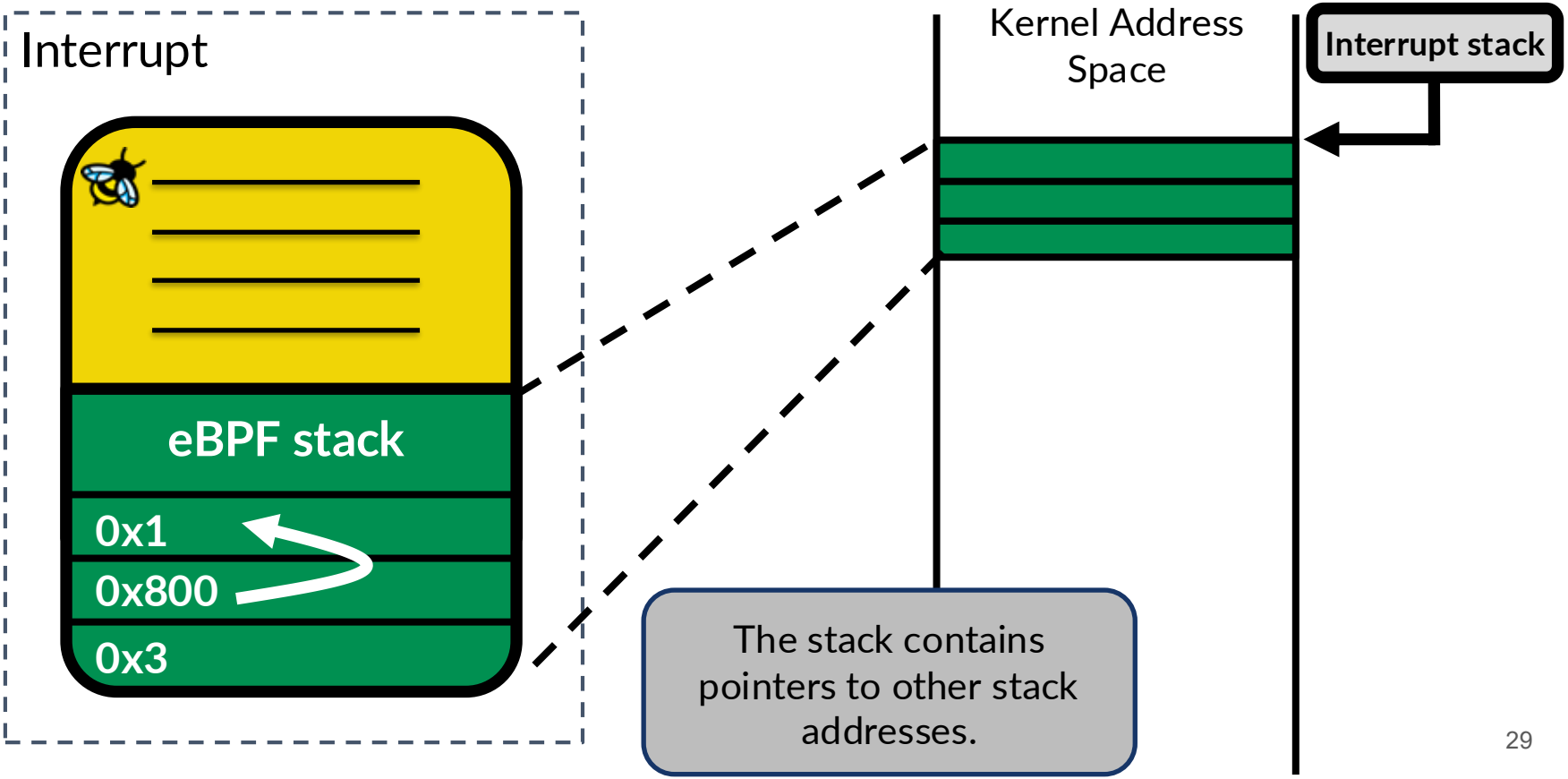
Solution: Utilize safety properties of helper functions



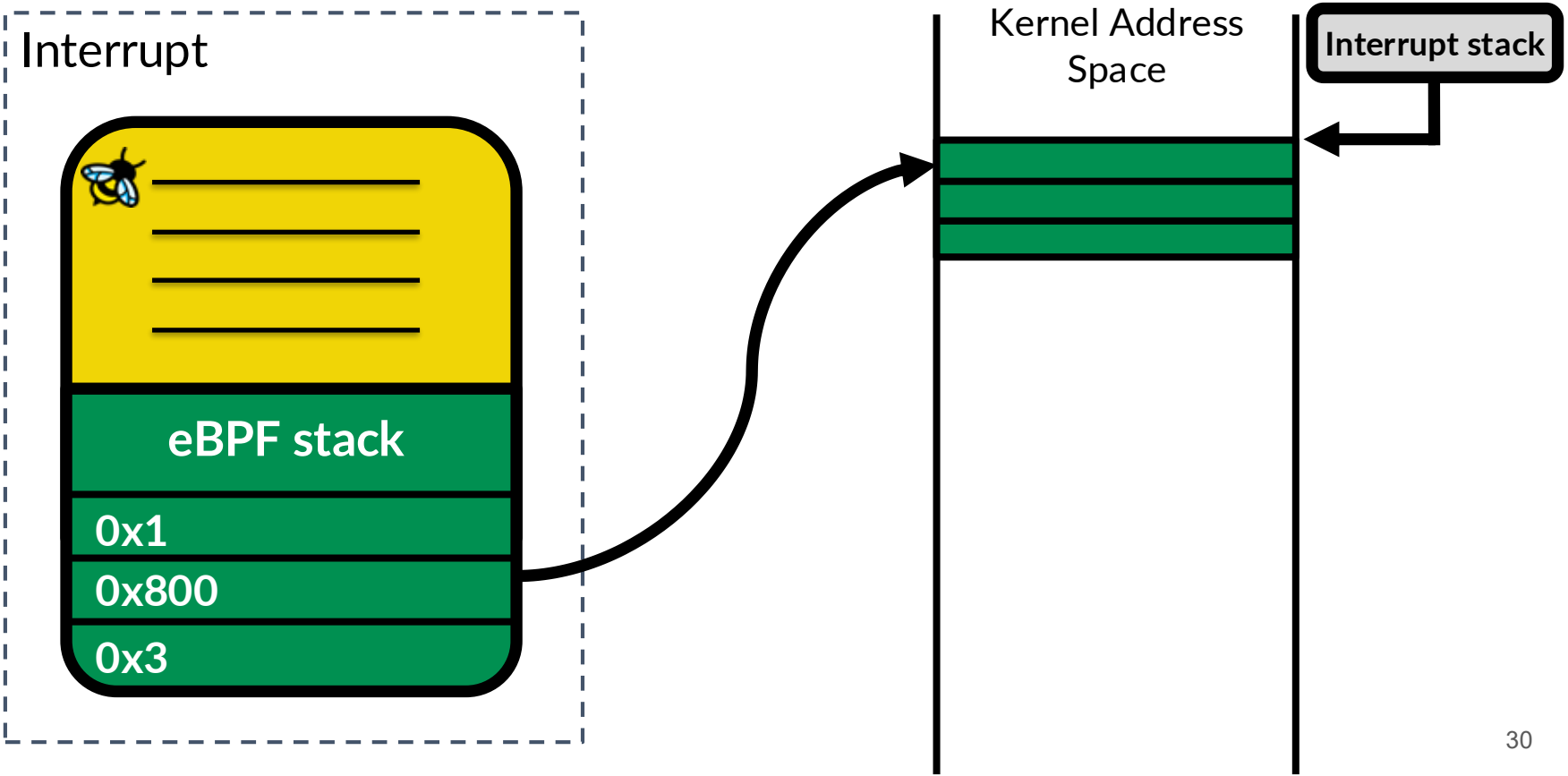
Solution: Utilize safety properties of helper functions



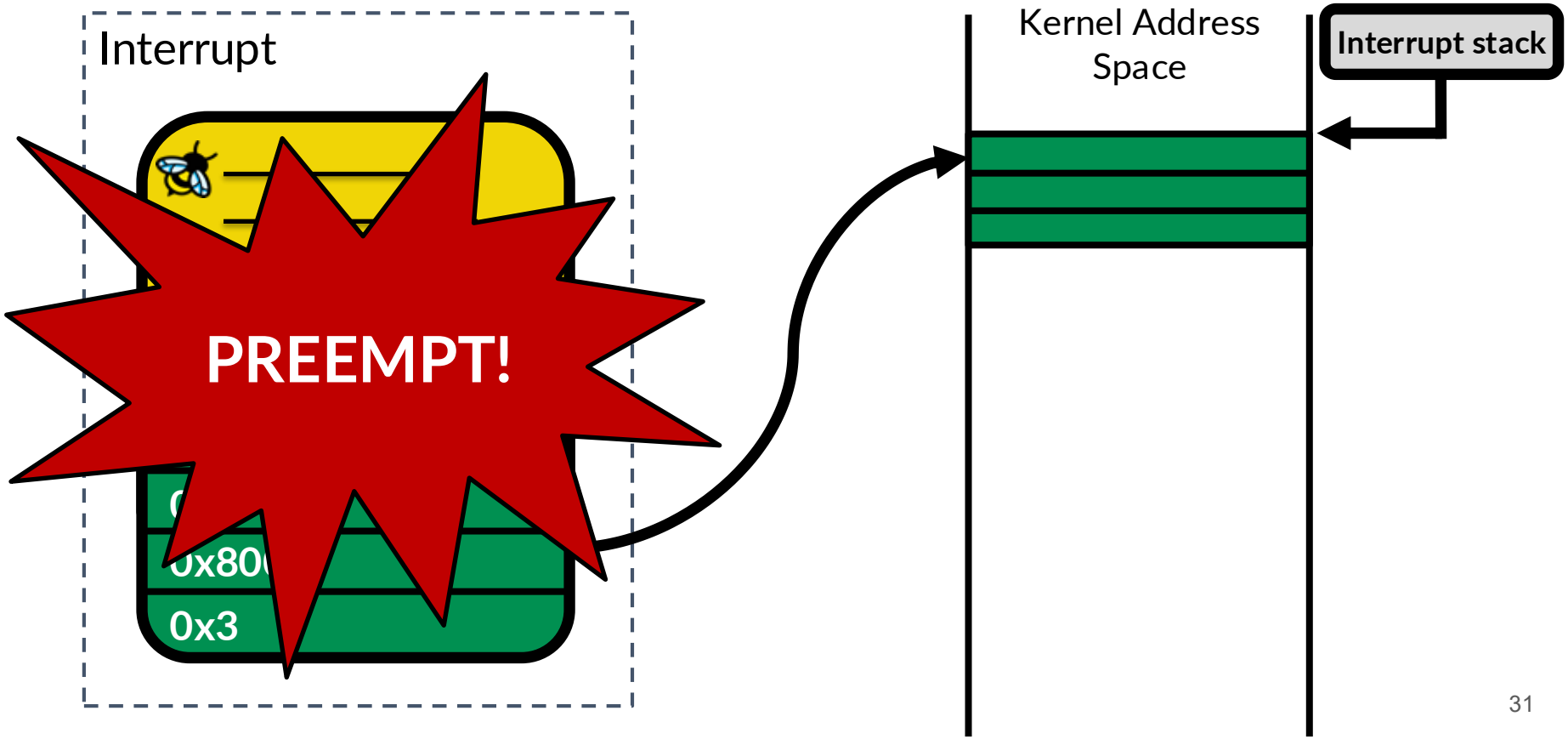
Challenge 2: Pointers must change on resumption



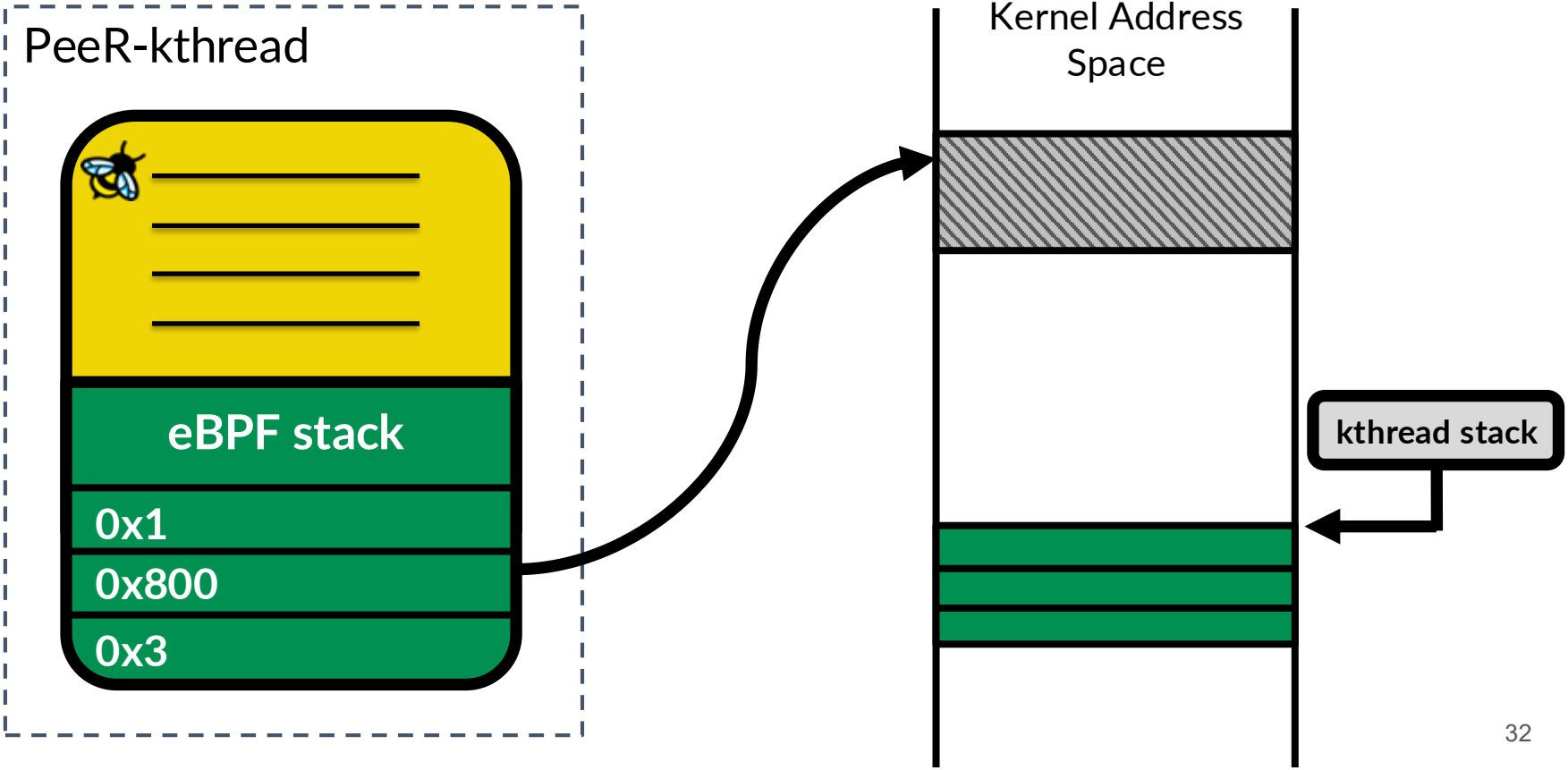
Challenge 2: Pointers must change on resumption



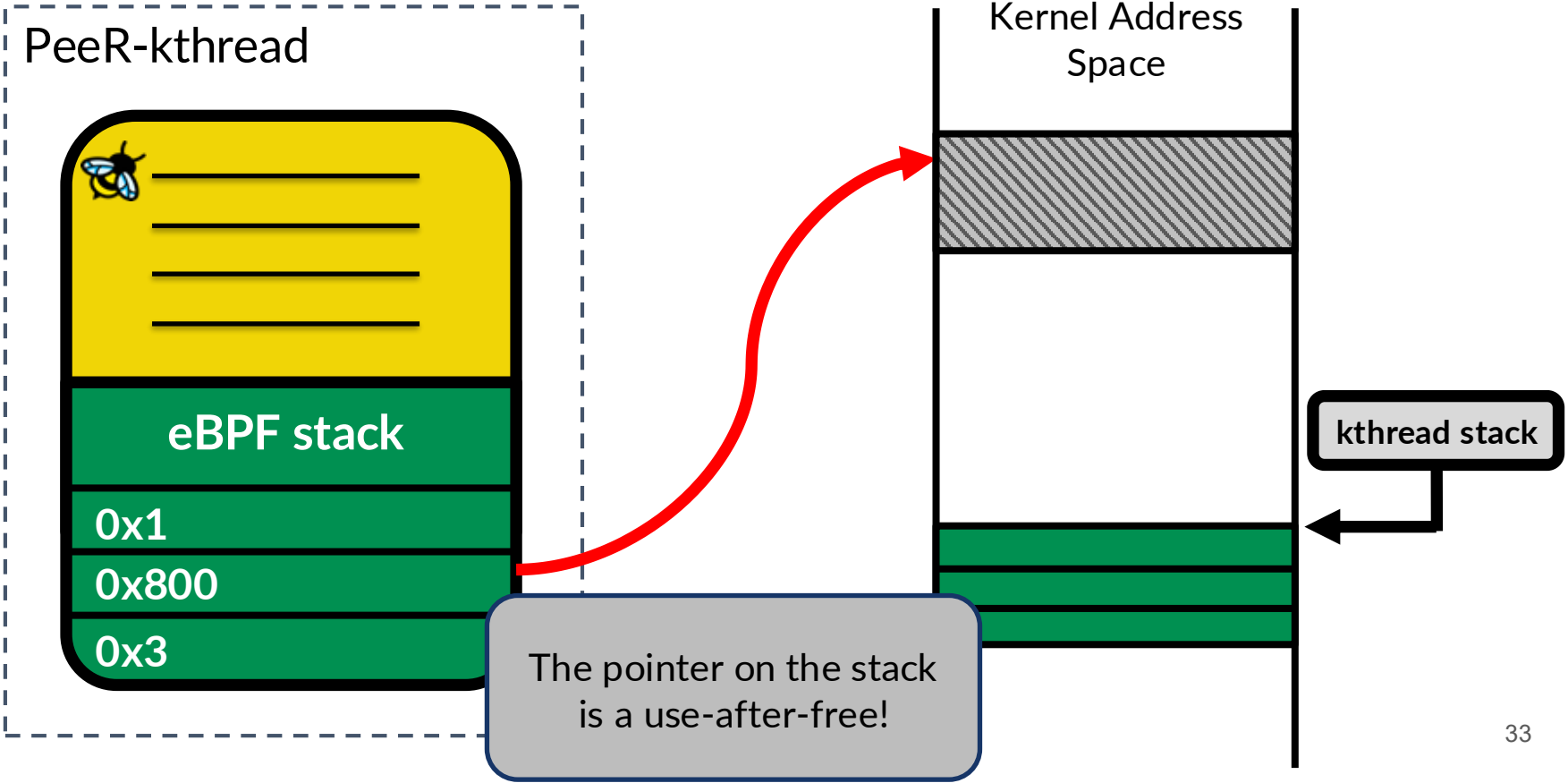
Challenge 2: Pointers must change on resumption



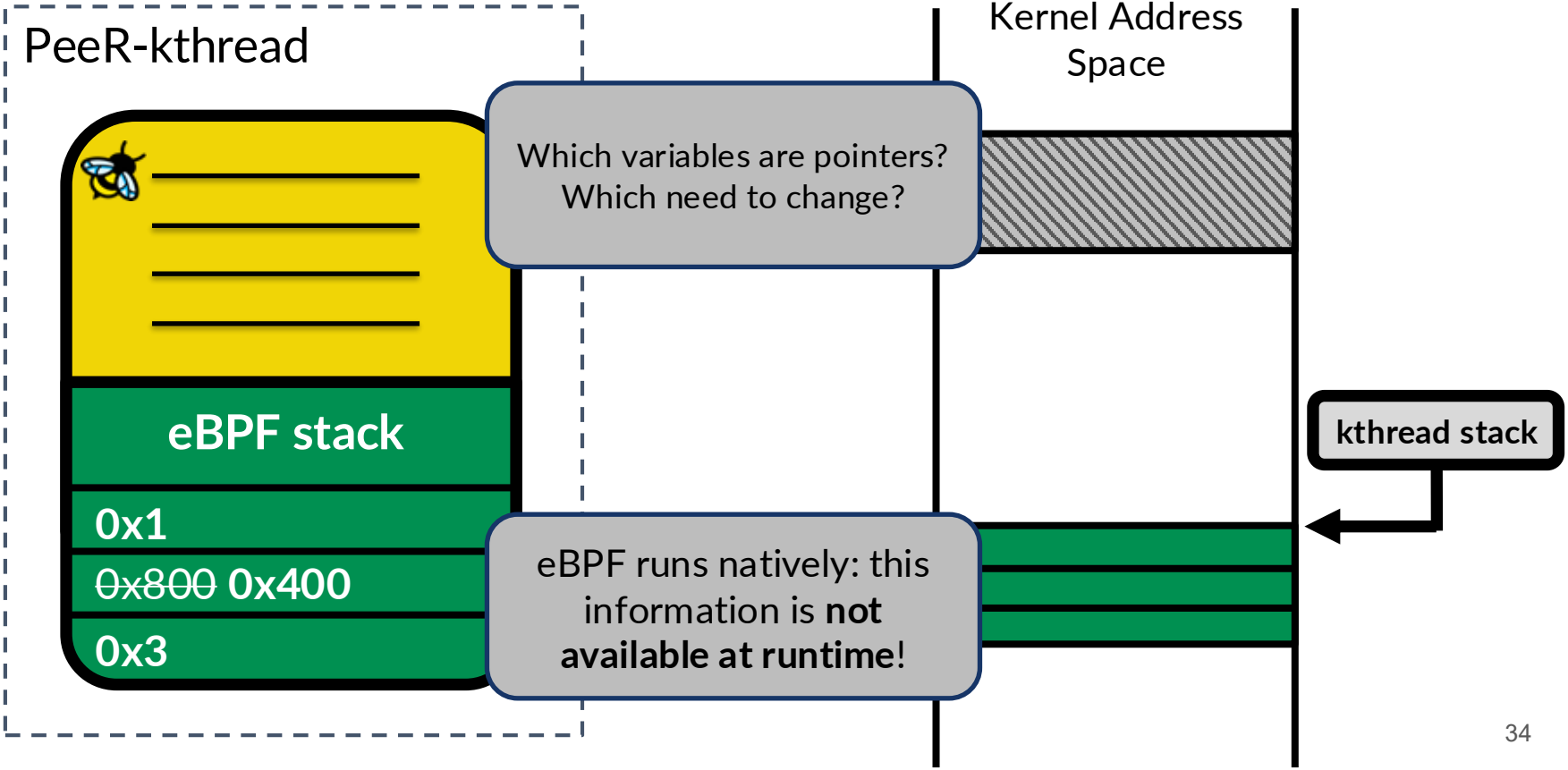
Challenge 2: Pointers must change on resumption



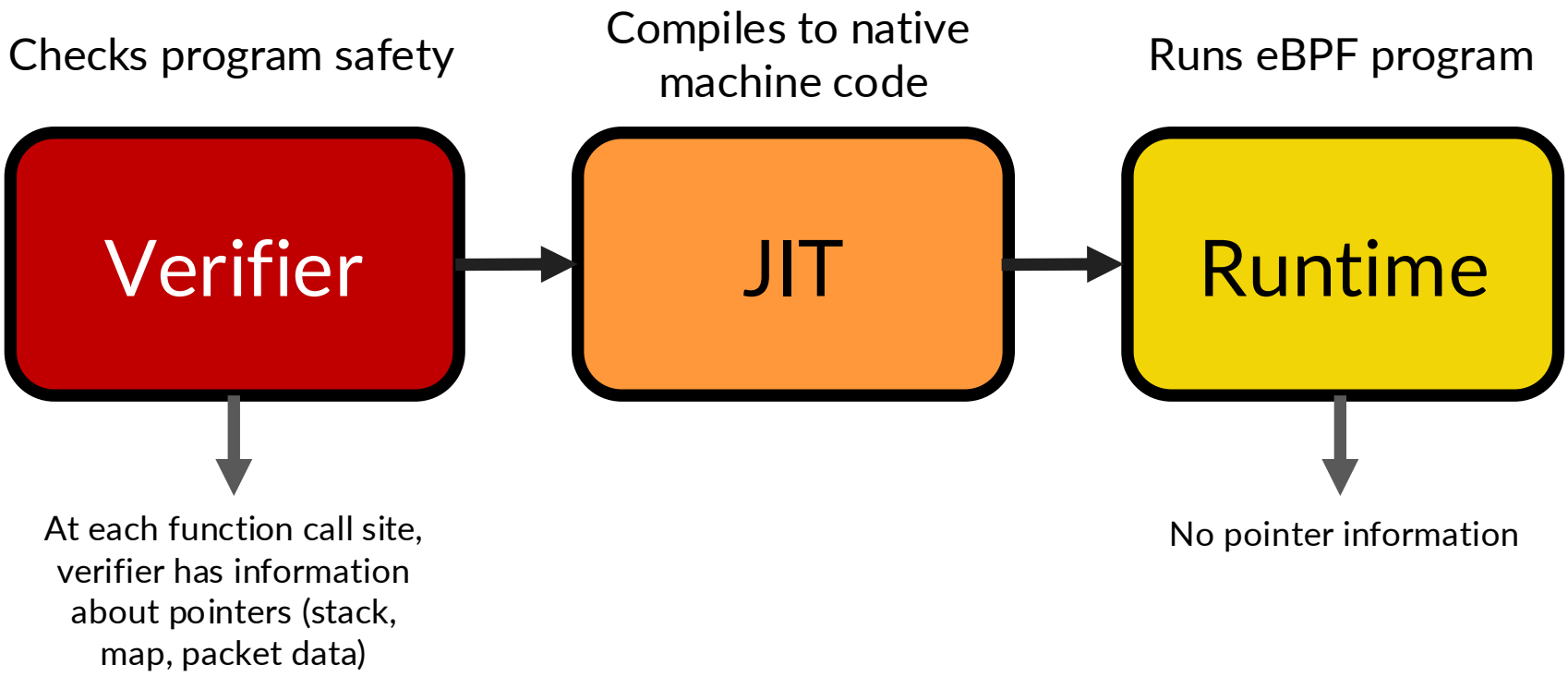
Challenge 2: Pointers must change on resumption



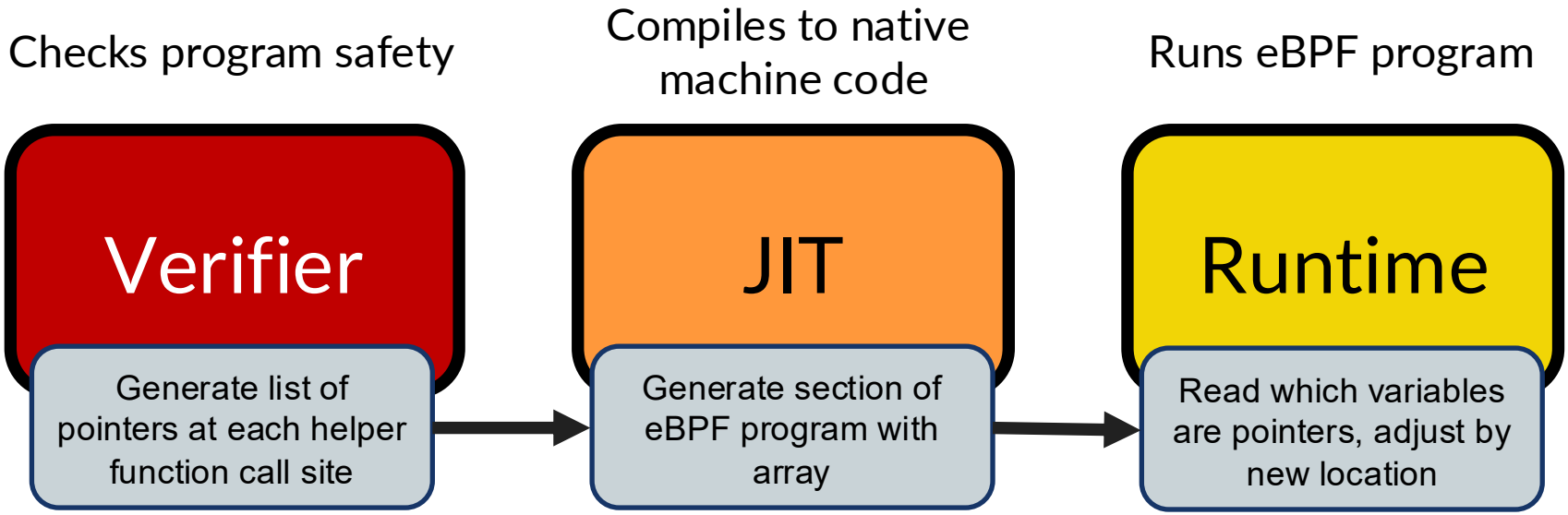
Challenge 2: Pointers must change on resumption



Solution: Use eBPF verifier information at runtime



Solution: Use eBPF verifier information at runtime



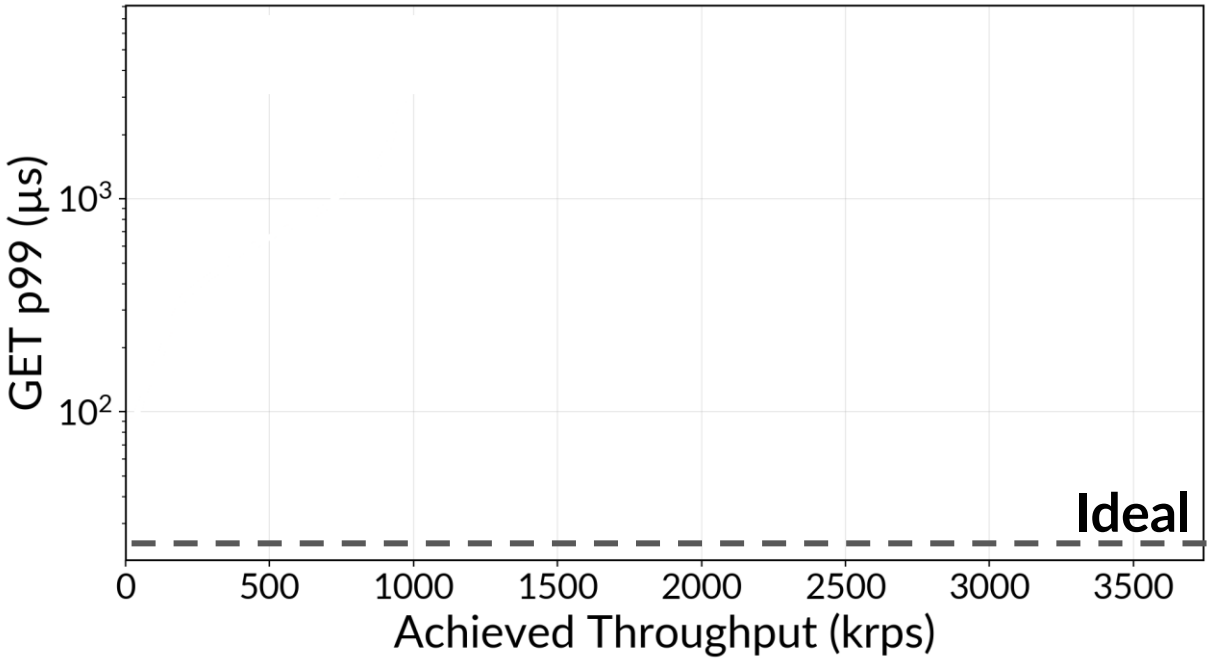
Does PeeR enable performance and colocation?

Evaluation setup

- Hardware and software:
  - Two servers connected by a 400 Gbps NIC
  - Server: KeyDB (multi-core Redis) in userspace, KFlex-Redis in eBPF (KFlex, SOSP ‘24)
  - Client: open-loop Redis client

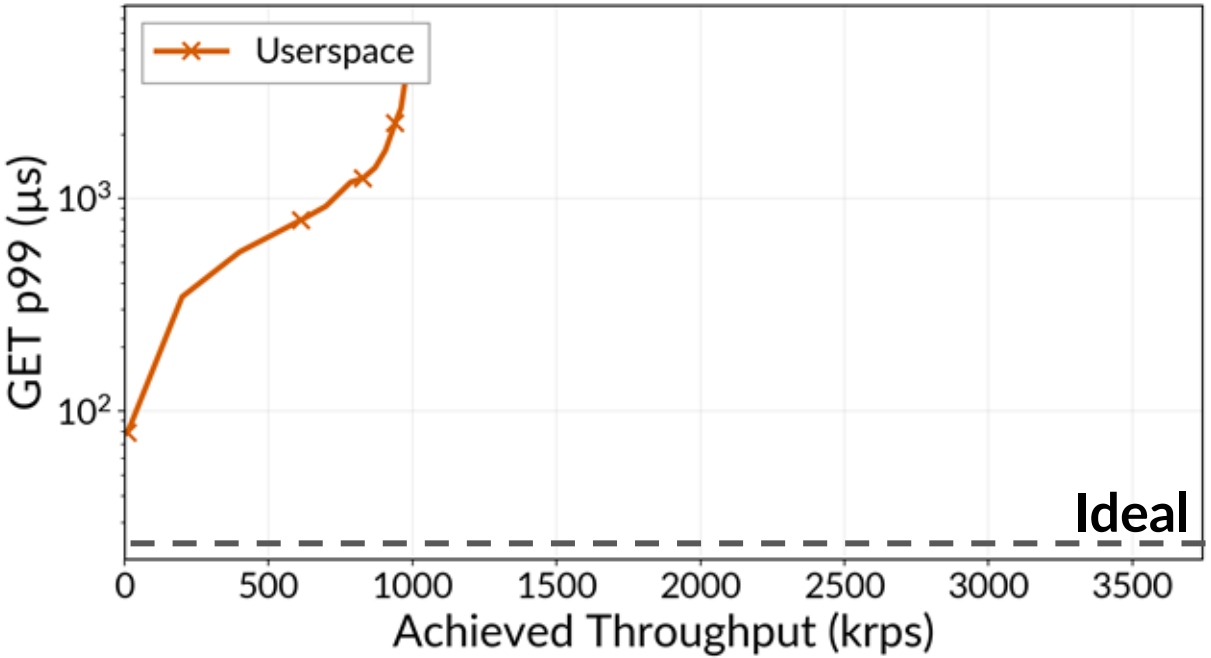
- Performance: point/scan mix: 99.5% GET (0.5 µs), 0.5% SCAN (200 µs)

Existing runtimes hit a tail latency wall



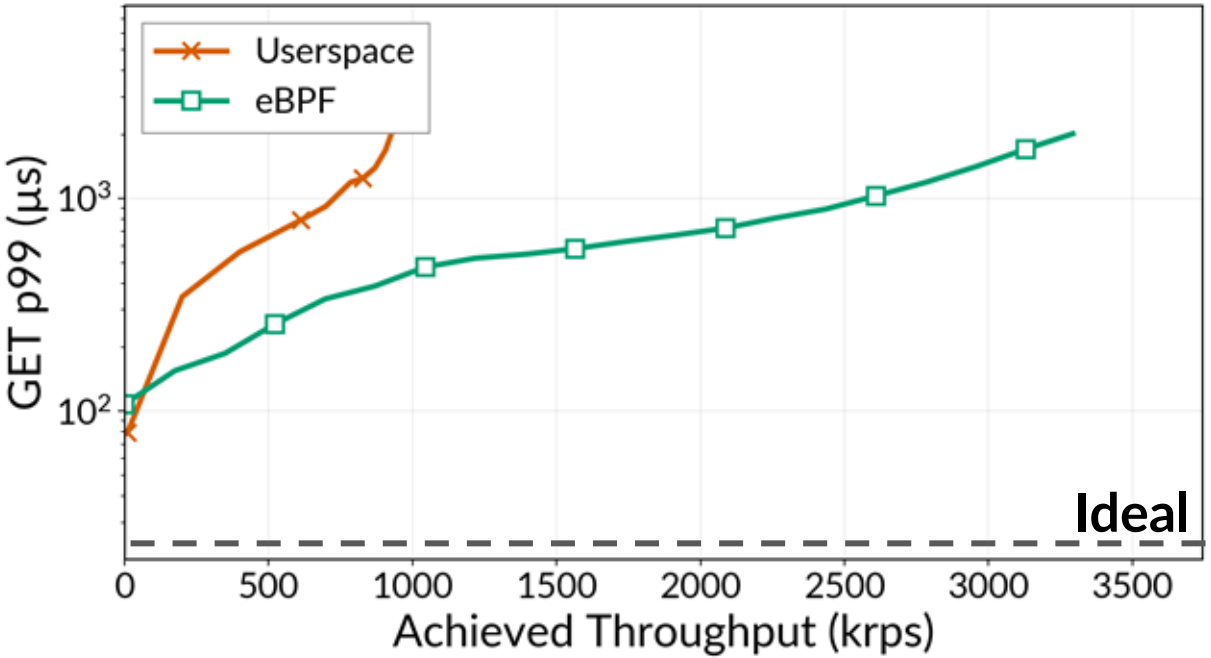
Ideal: only GET requests

Existing runtimes hit a tail latency wall



Userspace saturates at ~1M rps

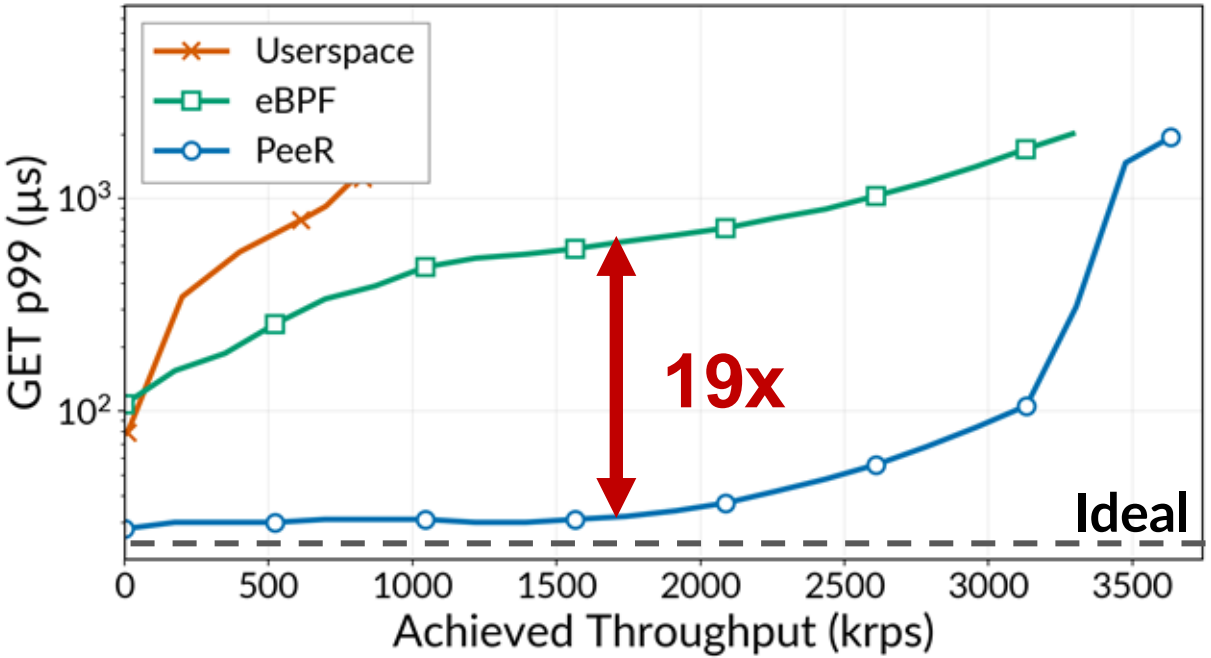
Existing runtimes hit a tail latency wall



Userspace saturates at ~1M rps

eBPF scales further, but p99 is far above ideal

PeeR eliminates head-of-line blocking



19x lower p99 GET latency

3.5x more throughput than userspace

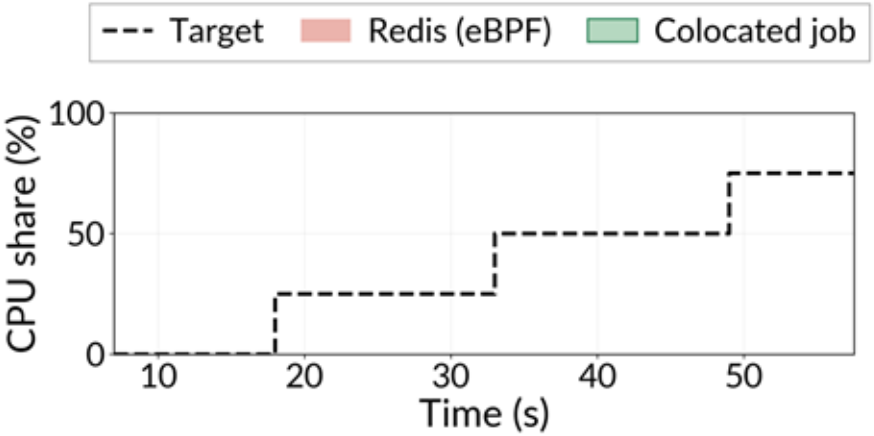
PeeR enables performance and colocation

- Setup:
  - Two servers connected by a 400 Gbps NIC
  - Server: KeyDB (multi-core Redis) in userspace, KFlex-Redis in eBPF (KFlex, Dwivedi et al.)
  - Client: open-loop Redis client

- Performance: point/scan mix: 99.5% GET (0.5 µs), 0.5% SCAN (200 µs)
  - Measure p99 of GET requests

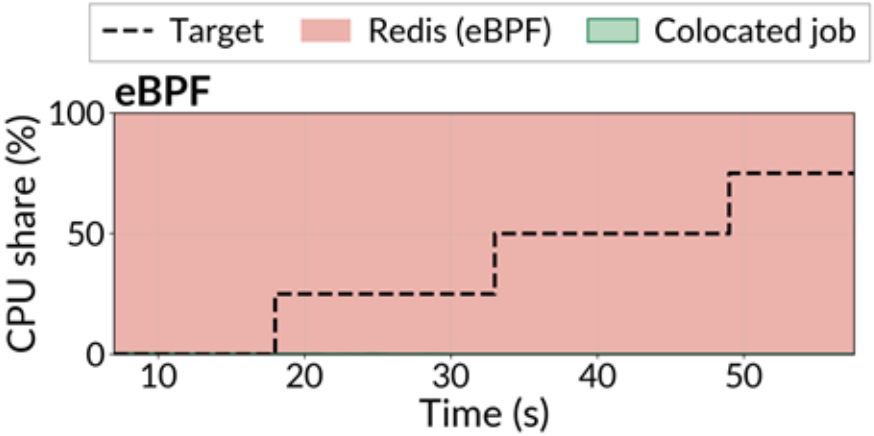
- Colocation: eBPF application and colocated userspace job
  - Over time, vary the % of CPU time dedicated to the colocated userspace job

eBPF is invisible to the scheduler



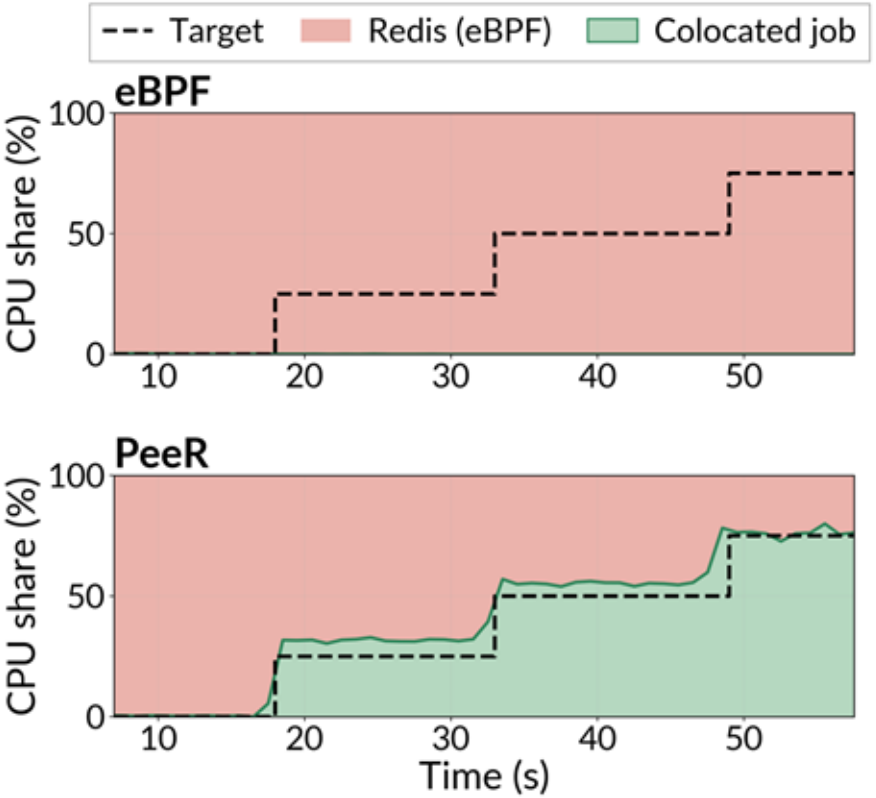
Operator varies the colocated job's CPU share

eBPF is invisible to the scheduler



Colocated application starves,
Redis runs at full throughput

PeeR enforces CPU shares



eBPF applications use only their allocated CPU

Colocated application gets CPU share

PeeR enables modern eBPF applications

- PeeR makes eBPF a viable runtime for full, latency-critical applications
- eBPF applications become first-class citizens for the scheduler

Conclusion

- PeeR makes eBPF preemptible and schedulable, enabling performance and colocation
- Up to 19x lower tail latency and enforced CPU sharing, with minimal overhead
- Open source!
- More in the paper: sched_ext integration, application-informed scheduling policies, overhead microbenchmarks, correctness analysis…

Paper and code:



Jeremy Carin
jcarin@mit.edu