Syrup: User-Defined Scheduling Across the Stack

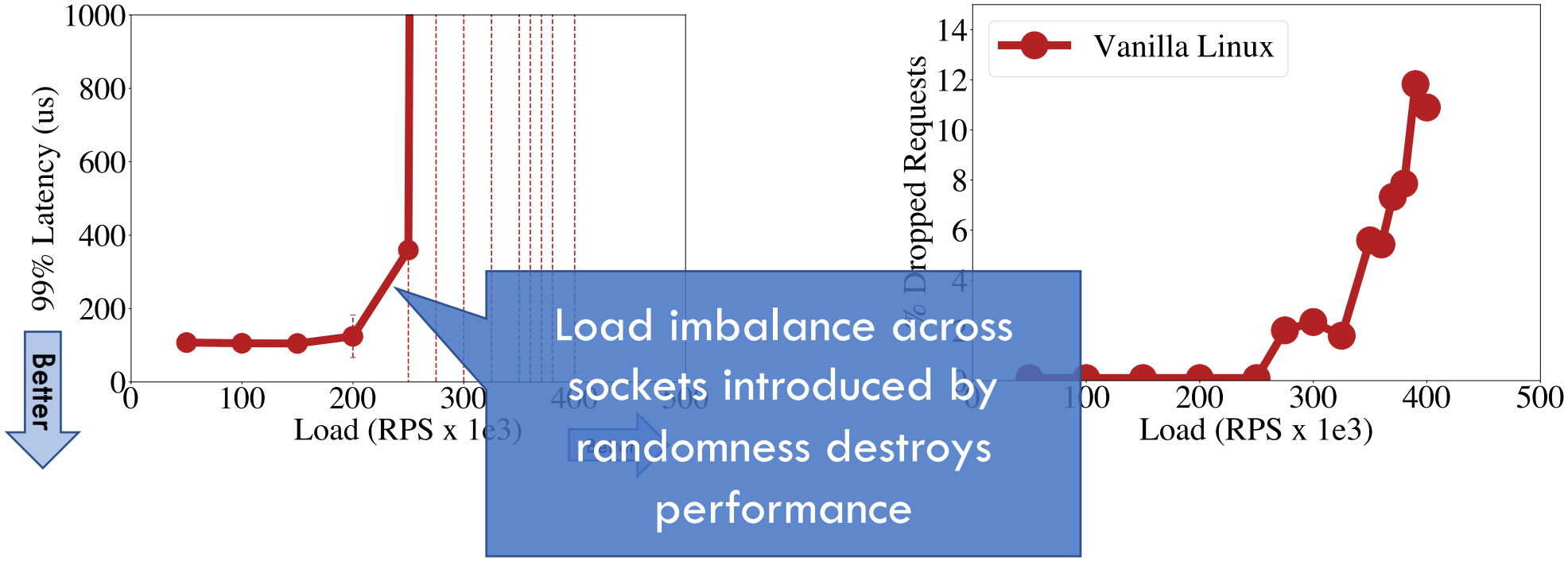
Kostis Kaffes, Jack Tigar Humphries,
David Mazières, Christos Kozyrakis

Scheduling matters

- "Good" scheduling eliminates problems such as ***head-of-line blocking***, ***lack of work conservation***, and ***load imbalance***.
- Fine-tailored policies can improve performance by an order of magnitude or more.

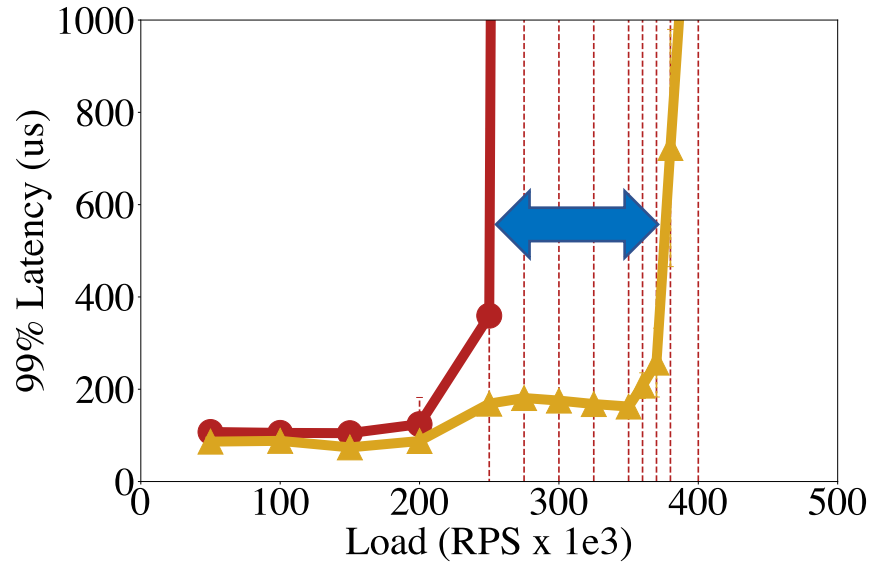
Example: RocksDB Server

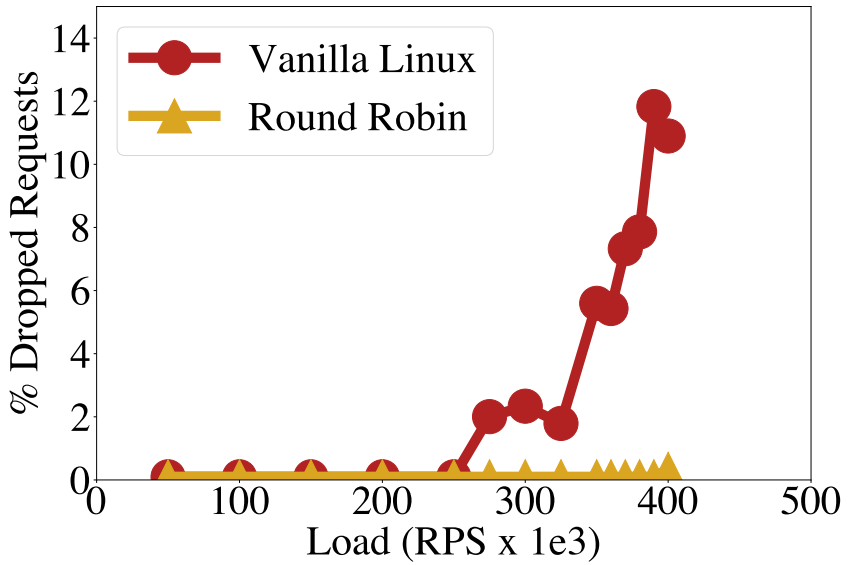
- Multi-threaded RocksDB UDP server using SO_REUSEPORT.
- Vanilla Linux assigns packets from 50 clients to socket/threads using a hash of the 5-tuple.



Example: RocksDB Server

- Round Robin iterates over sockets/threads.

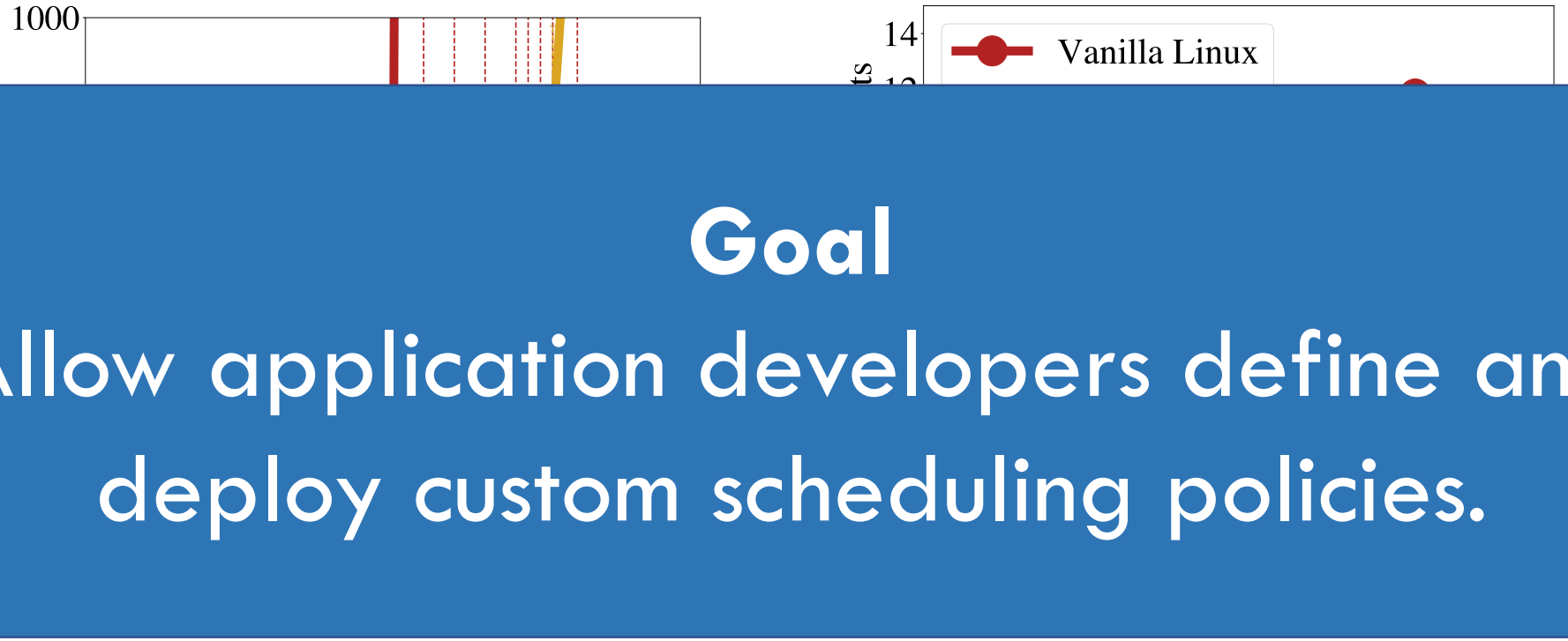




Simply changing socket matching policy provides **> 75%** performance improvement.

Example: RocksDB Server

- Round Robin iterates over sockets/threads.



Goal

Allow application developers define and deploy custom scheduling policies.

performance improvement.

Outline

- Motivation
- Requirements
- Syrup Design
- Evaluation
- Discussion

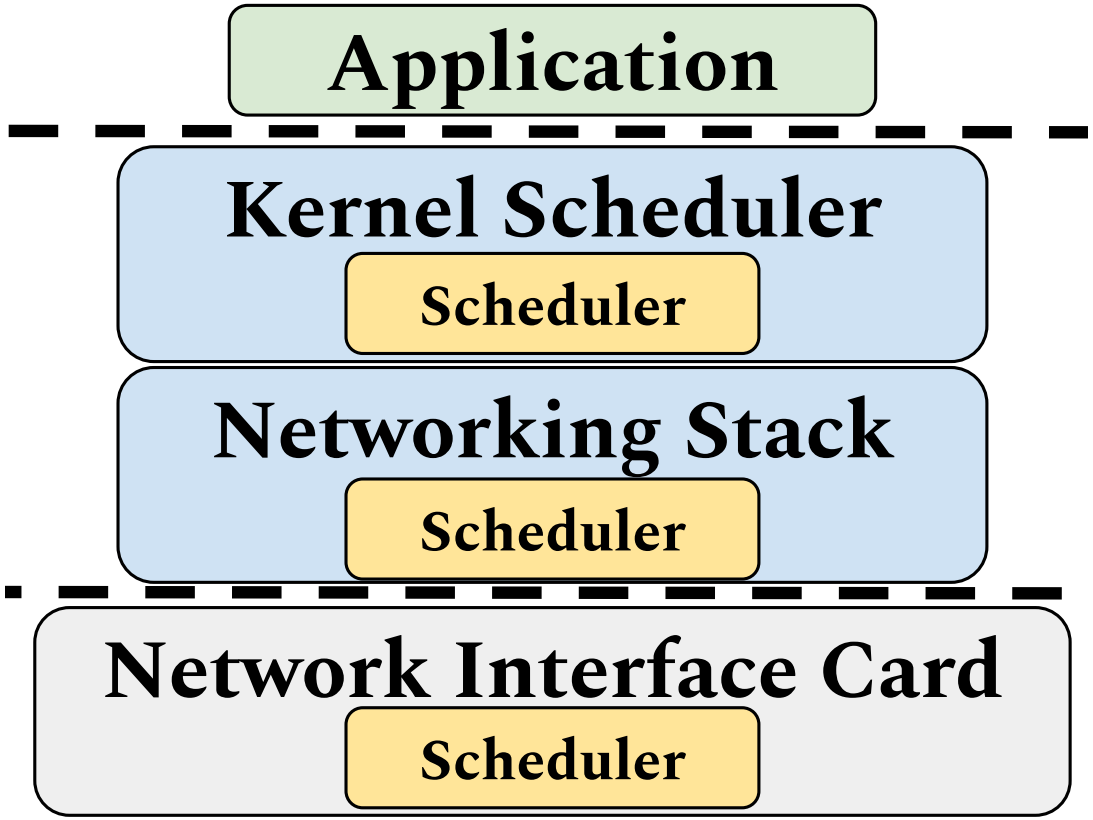
#1: **Expressibility**

Different applications and workloads perform best under different scheduling policies:

- Low Variability → FCFS scheduling
- High Variability → Preemption or Resource Partitioning
- Memory Intensive → Locality
- …

#2: **Cross-Layer Deployment**

Scheduling takes place across multiple layers of the stack.



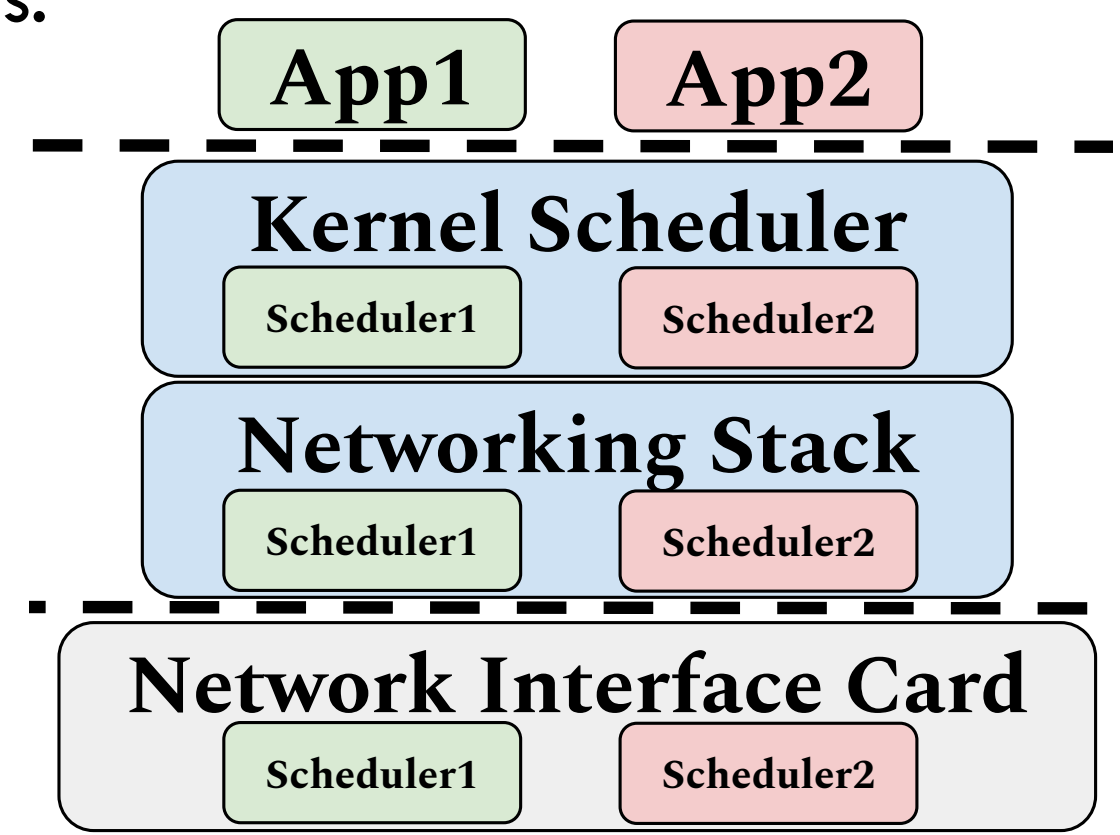
#3: **Low Overhead**

Many modern workloads operate at microsecond scale.

Making and **enforcing** a scheduling decision should not add too much overhead → ~1us or less!

#4: **Isolation**

Different applications should be able to safely co-deploy their custom policies.



“Legacy” Scheduling Options

Implement your favorite policy in the Linux kernel:

+ Can implement any policy.
- Hard to coordinate across layers.
- Probably only people attending netdev can do it.

Build a data-plane OS:

+ High performance / low overhead.
- Incompatible with existing applications.
- Hard and costly to maintain

Tools for Malleable Scheduling

eBPF

In-kernel virtual machine.

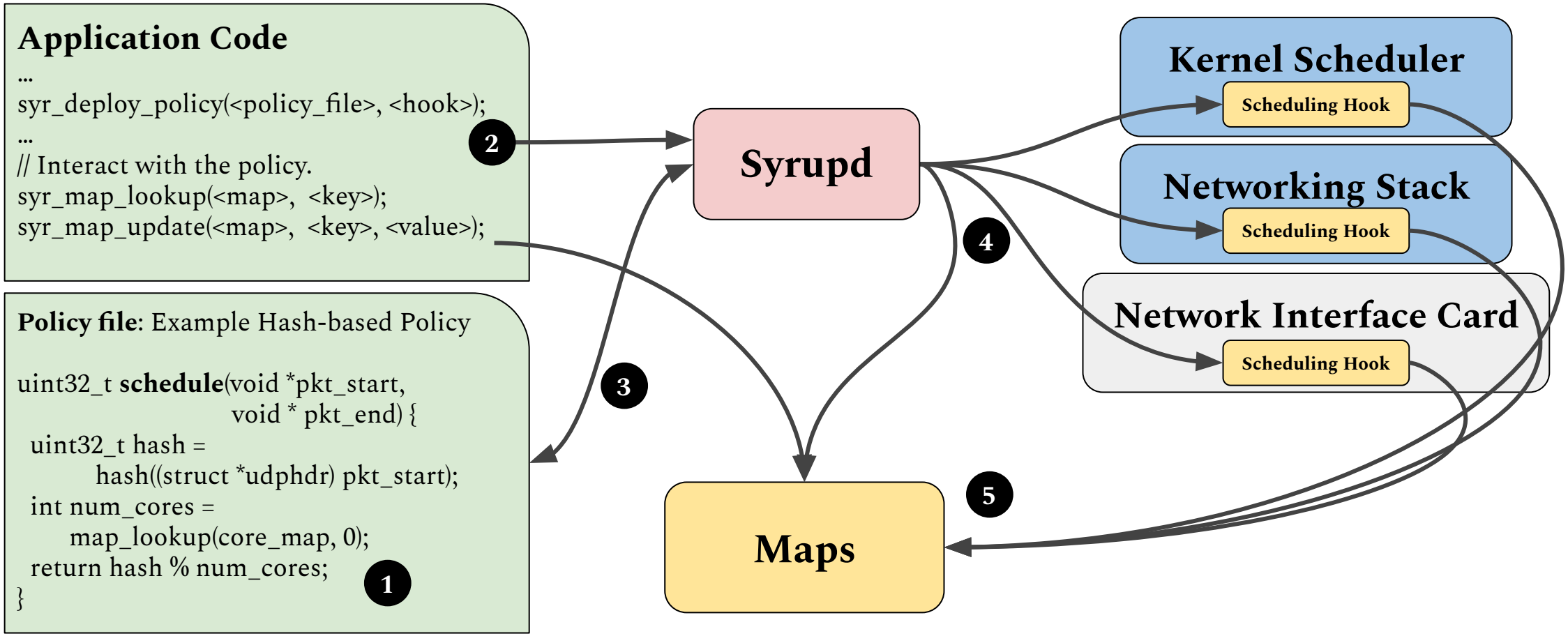
Allows running sanitized user-provided code in the kernel.

ghOSt

Framework that offloads kernel thread scheduling to userspace agents.

Outline

- Motivation
- Requirements
- Syrup Design
- Evaluation
- Discussion



1. Specify the scheduling policy in C.
2. Deploy the policy from the application code.
3. A daemon (Syrupd) compiles the policy for the target hook(s).
4. The daemon (Syrupd) deploys the policy to the target hook(s).
5. The different layers can communicate using eBPF maps.

Meeting the Scheduling Requirements

1. Expressibility → Treat scheduling as online matching problem.
2. Cross-layer deployment → Leverage eBPF and ghOSt mechanisms to deploy code across the stack.
3. Low overhead → See evaluation.
4. Isolation → Use a global arbiter that manages scheduling policies for different applications.

Scheduling as Online Matching

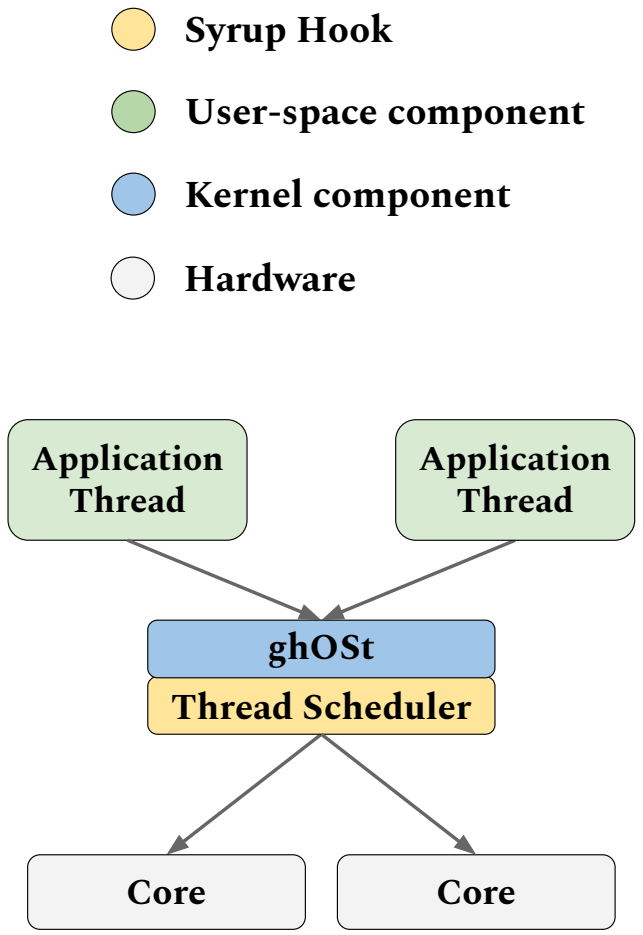
Syrups represents scheduling policies as matching functions between inputs and executors that process the inputs.
Inputs: Network packets, connections, threads, …
Executors: NIC queues, network sockets, cores, …

+ Almost declarative scheduling – Users specify the matching and the underlying system enforces it.
+ Scheduling code portable across different layers of the stack.
+ Scheduling broken down into a series of "small" decisions, improving the composability and the understandability of even complex policies.

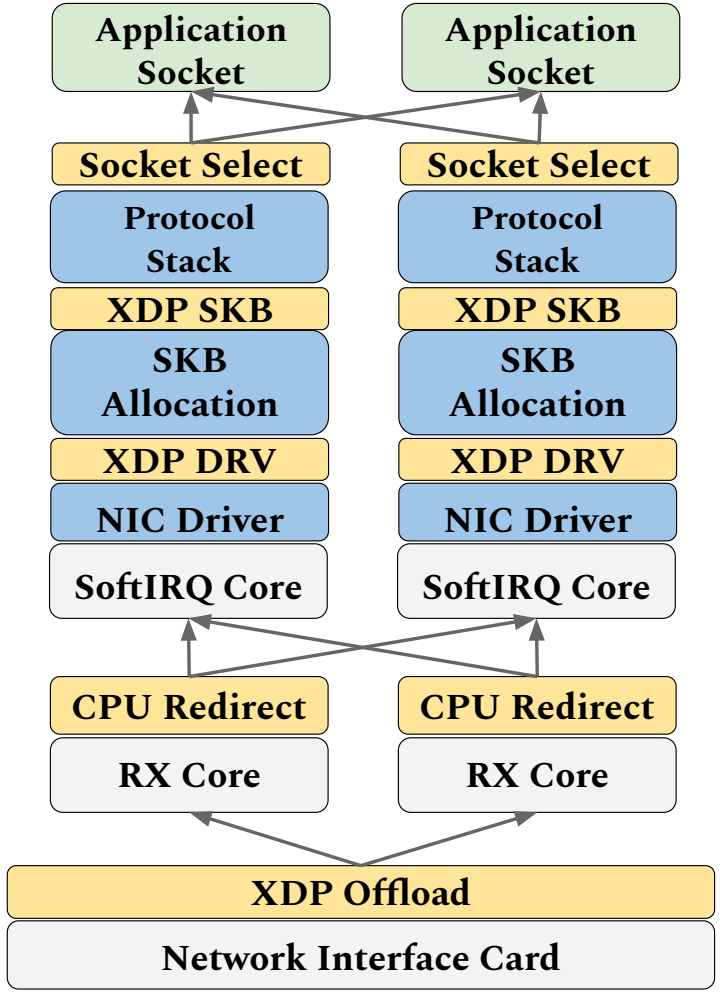
Policy Example: Round-Robin Thread Selection

```
1 uint32_t idx = 0;
2 uint32_t schedule(void *pkt_start,
3                    void *pkt_end) {
4   idx++;
5   return idx % NUM_THREADS;
6 }
7
```

Scheduling across the stack with eBPF and ghOSt



Thread Scheduling Syrup Hook



Network Stack Syrup Hooks

Cross-layer communication with eBPF maps

User-defined eBPF maps are used to communicate between different scheduling hooks and the user-space.

System-defined eBPF maps are used to hold references to executors, e.g., network sockets or cores.

Providing Isolation Between the Kernel and Applications

The eBPF verifier makes sure that an application policy does "break" the underlying system.

ghOSt scheduling policies run at lower priority than CFS, allowing the system to reclaim resources.

Syrupd Provides Isolation Among Applications

All policy deployment requests go through Syrupd which:

1. Uses BPF_MAP_TYPE_PROG_ARRAY to filter inputs to application-specific policies in eBPF hooks.
2. Deploys ghOSt user-space agents for each application that only handle the corresponding application's threads.

Outline

- Motivation
- Requirements
- Syrup Design
- Evaluation
- Discussion

Evaluation Questions

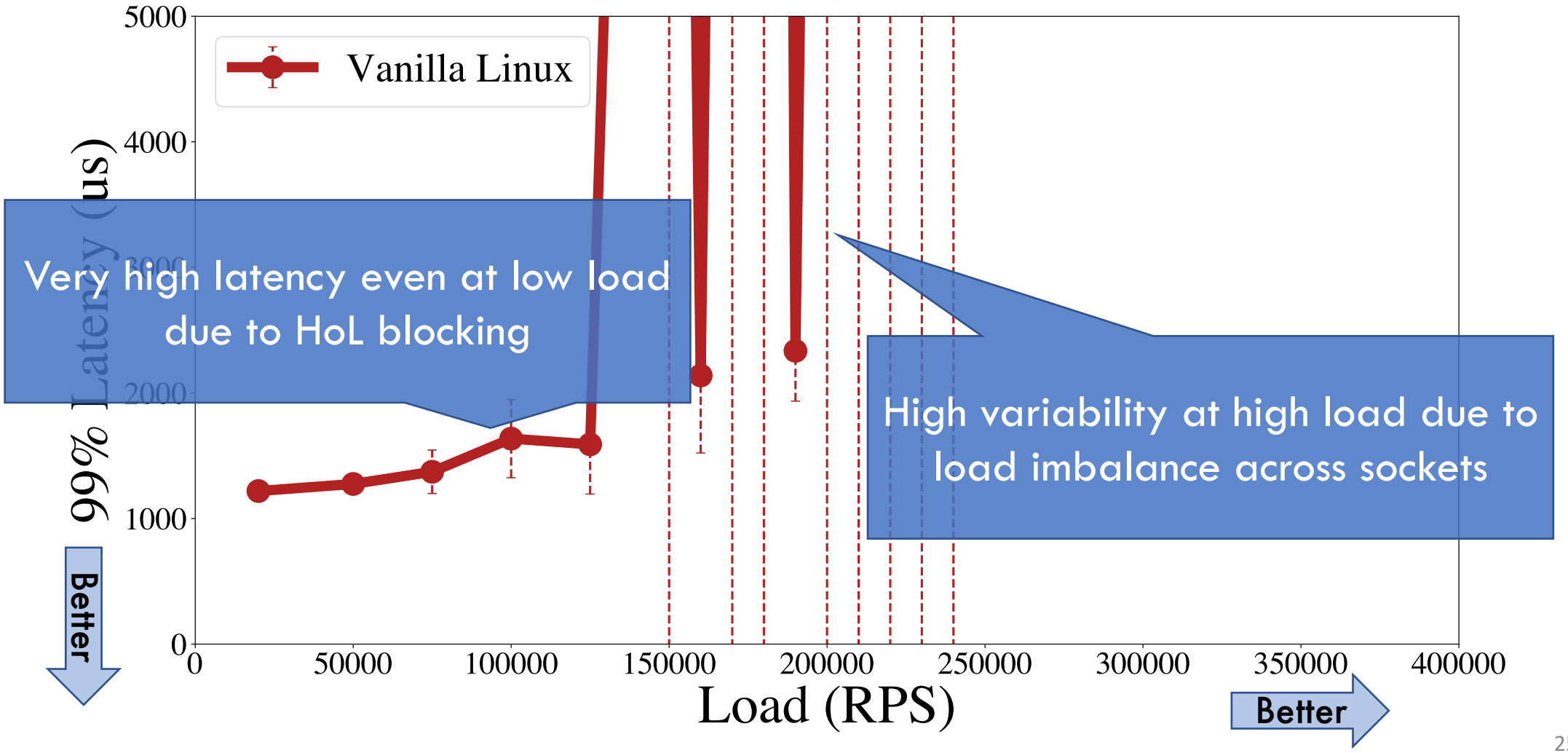
1. Can Syrup be used to express and implement a variety of scheduling policies?
2. Can Syrup be used for cross-layer scheduling?
3. What are Syrup's overheads?

Experimental Setup

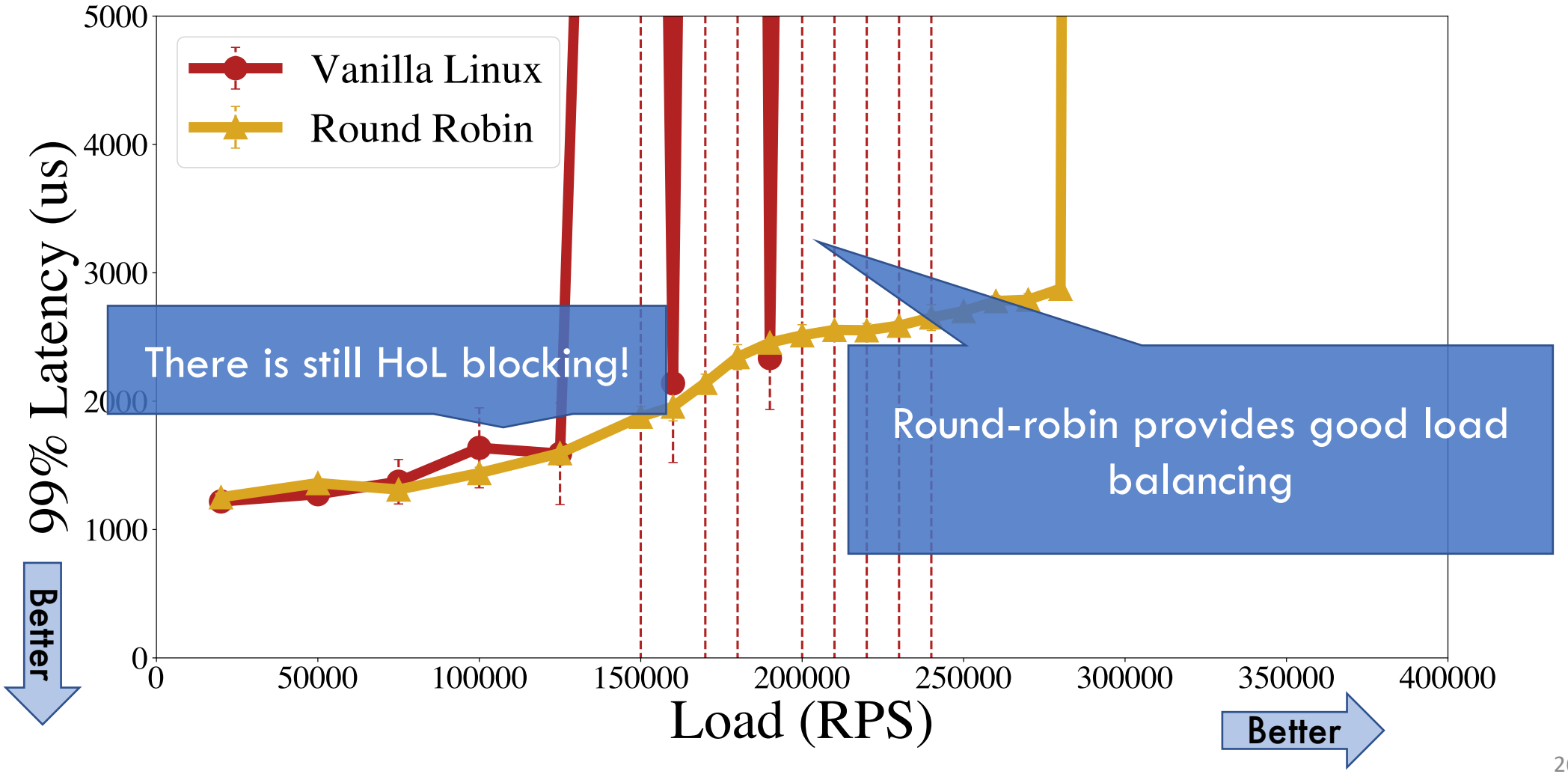
→Multi-threaded **RocksDB** UDP server using SO_REUSEPORT.
→Each thread pinned to a different core.
→Serving mix of 99.5% **GET**s (10 usec) and 0.5% **SCAN**s (700 us).

We use to Syrup to implement various socket selection policies.

Vanilla Linux Policy



Round-Robin Policy



SCAN Avoid Policy

USERSPACE

```
1 Request * req = parse_request(pkt);
2 if (req->type == SCAN)
3   map_update(&scan_map, &tid, SCAN);
4 // Do processing...
5 if (req->type == SCAN)
6   map_update(&scan_map, &tid, GET);
7
```

Notify kernel when handling a SCAN.

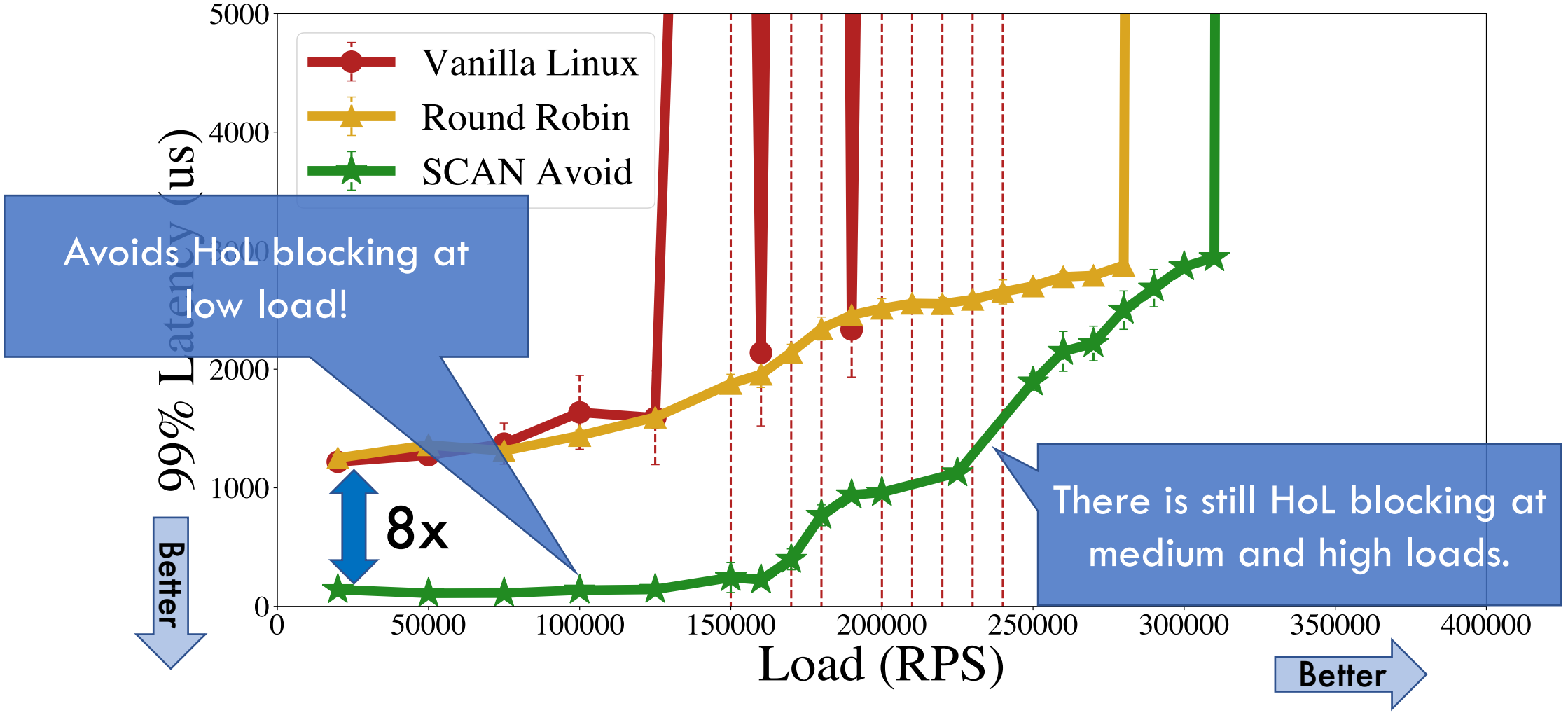
SCAN Avoid Policy

Avoid scheduling packets to cores handling SCANs.

KERNEL

```
1  uint32_t schedule(void *pkt_start,
2                    void *pkt_end) {
3    uint32_t cur_idx = 0;
4    for (int i = 0; i < NUM_THREADS; i++) {
5      cur_idx = get_random() % NUM_THREADS;
6      uint64_t * scan = map_lookup(&scan_map, &cur_idx)
         ;
7      if (!scan)
8        return PASS;
9      // Stop searching when a non-SCAN core is found.
10     if (*scan == GET)
11       break;
12   }
13   return cur_idx;
14 }
15
```

SCAN Avoid Policy



Size Interval Task Assignment Policy

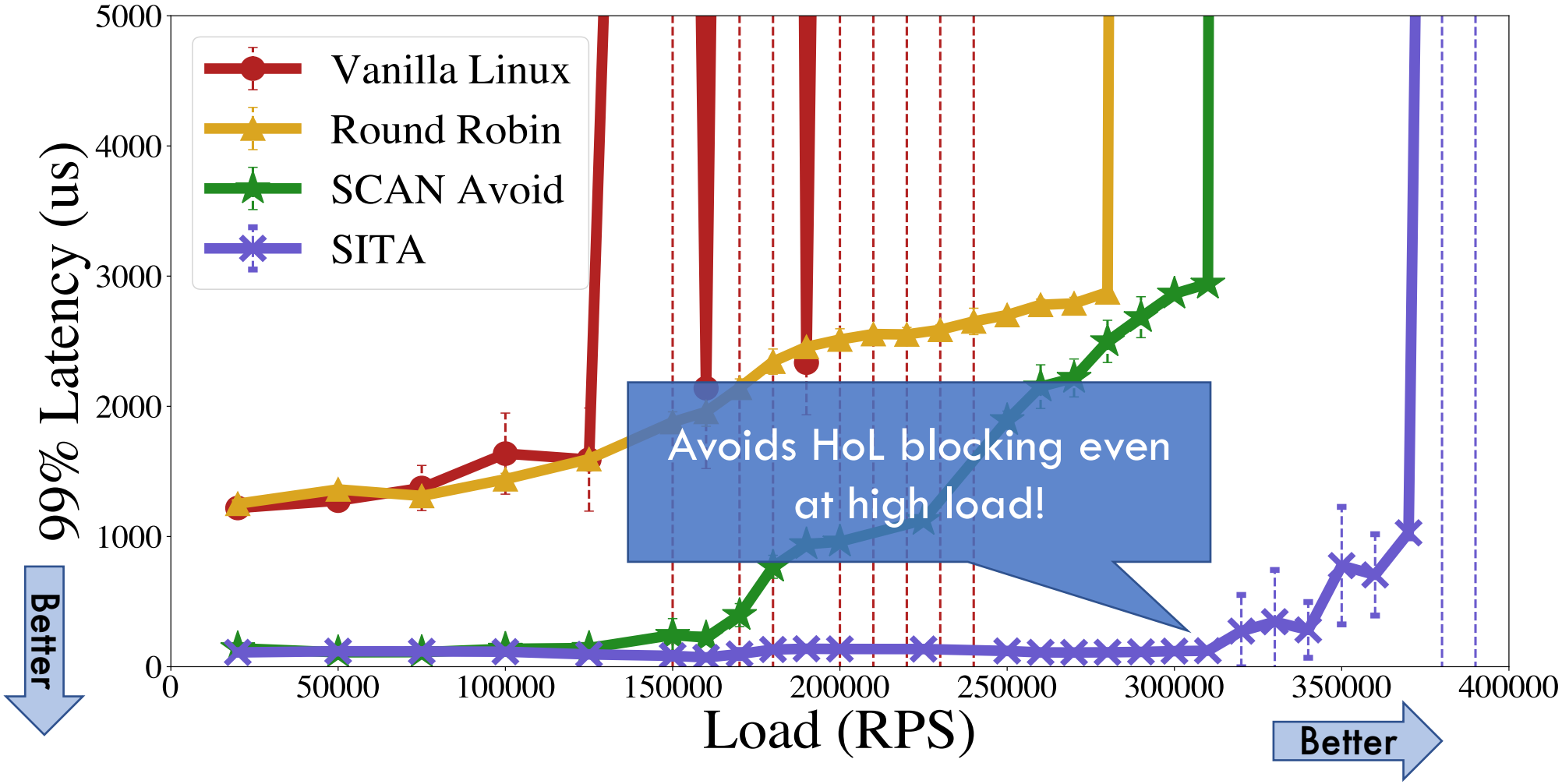
Parse the request type in the kernel.

Steer all SCANs to a specific thread.

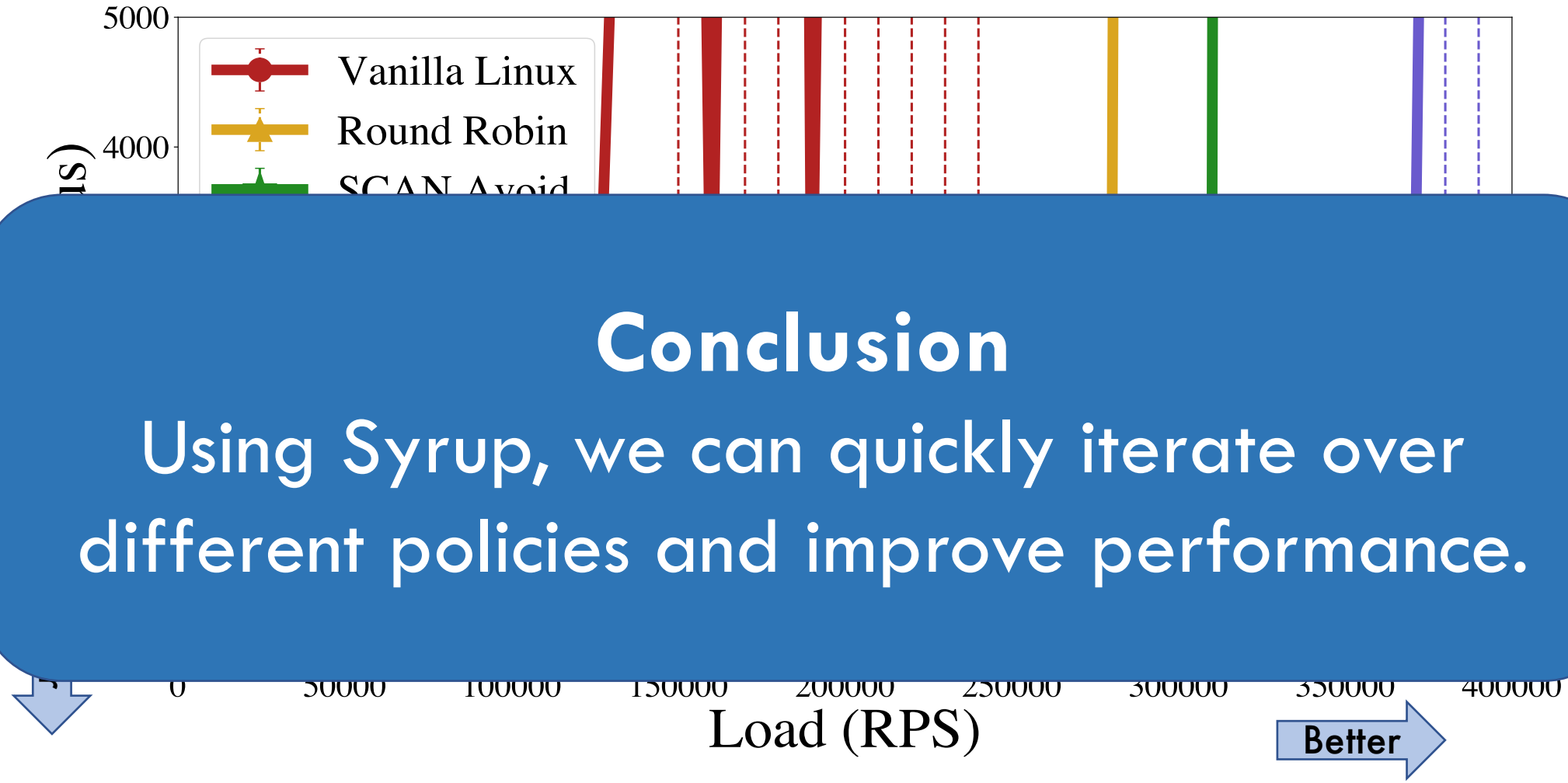
KERNEL

```
1  uint32_t idx = 0;
2
3  uint32_t schedule(void *pkt_start,
4                    void *pkt_end) {
5    if (pkt_end - pkt_start < 16)
6      return PASS;
7
8    // First 8 bytes are UDP header.
9    uint64_t * type = (uint64_t *)(pkt + 8);
10
11   if (*type == SCAN)
12     return 0;
13
14   idx++;
15   return (idx % (NUM_THREADS - 1)) + 1;
16 }
17
```

SITA Policy


Vanilla Linux
Round Robin
SCAN Avoid
SITA
99% Latency (us)
5000
4000
3000
2000
1000
0
0
50000
100000
150000
200000
250000
300000
350000
400000
Load (RPS)
Avoids HoL blocking even at high load!
Better
Better

SITA Policy



Conclusion

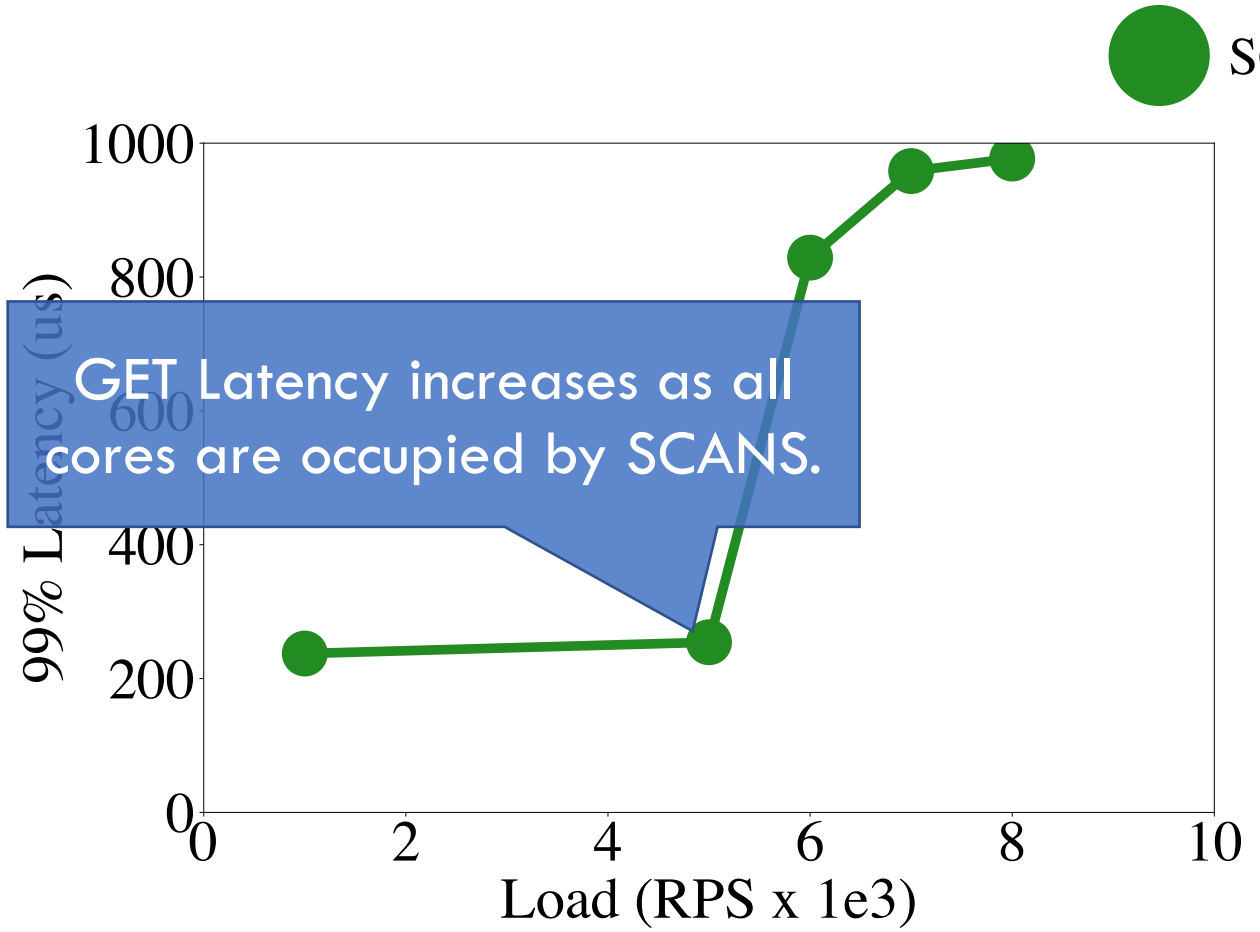
Using Syrup, we can quickly iterate over different policies and improve performance.

Scheduling Across Layers

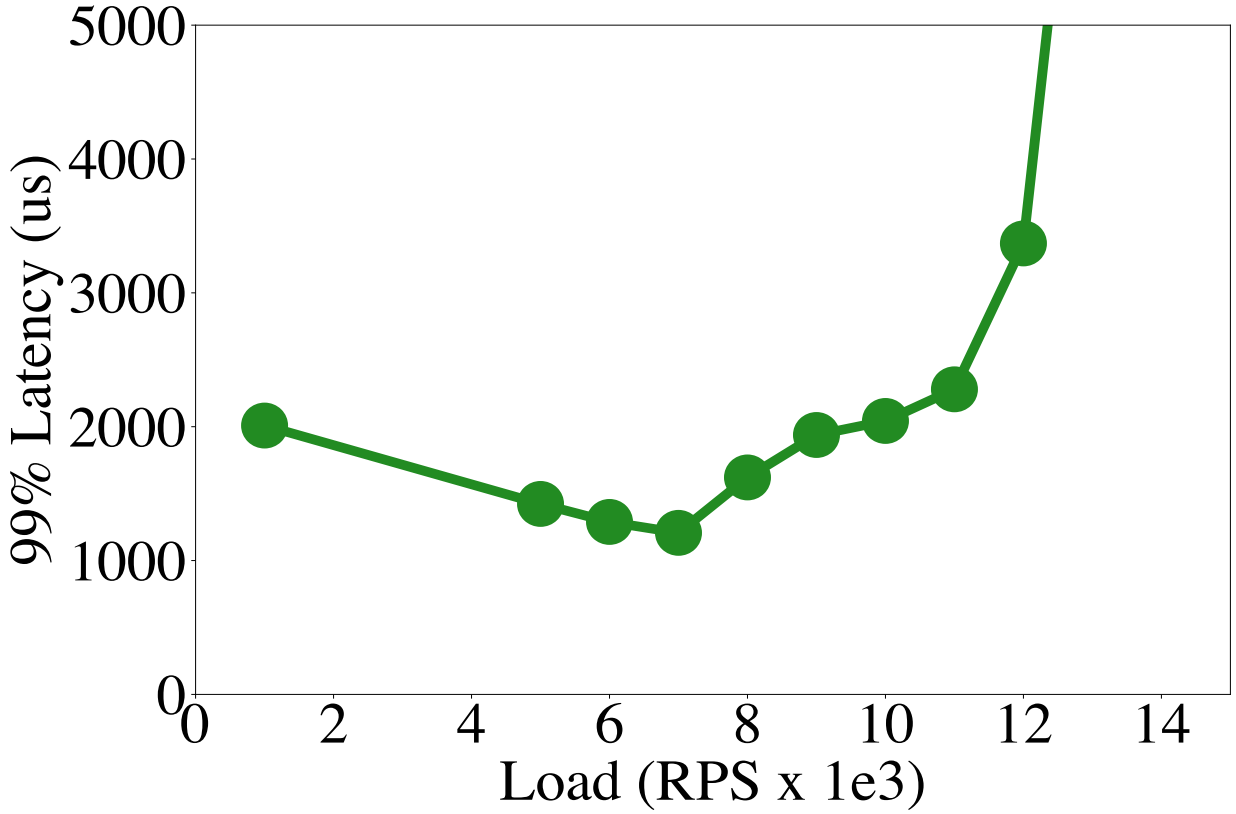
RocksDB workload 50% GETs -- 50% SCANs.

Problem: Most of the load comes from SCANs.
Solution: Use the SCAN Avoid policy and add more threads to avoid HoL blocking.

SCAN Avoid – 50% GET – 50 % SCAN



GET Latency



SCAN Latency

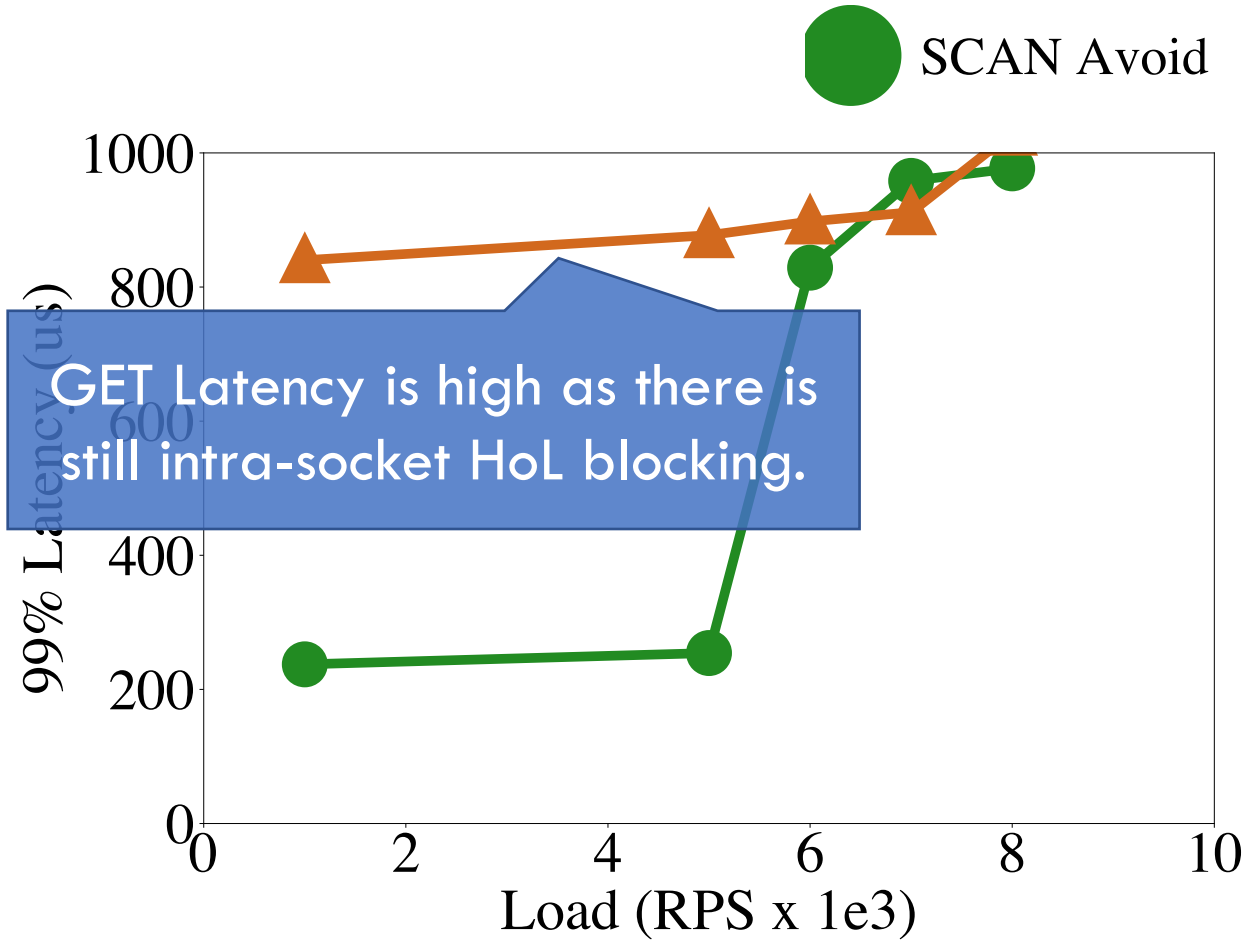
Scheduling Across Layers

RocksDB workload 50% GETs -- 50% SCANs.

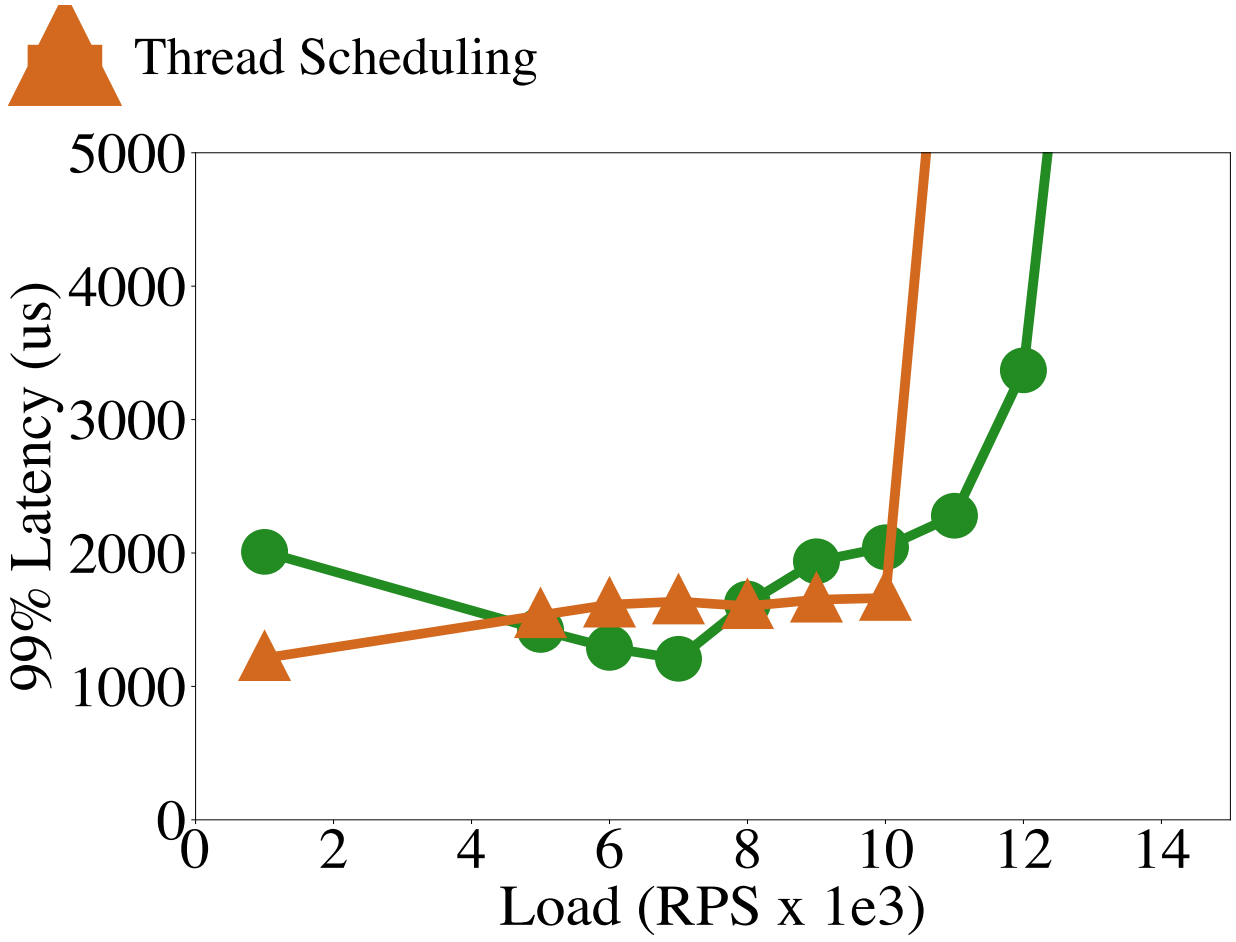
Problem: Most of the load comes from SCANs.
~~Solution: Add more threads to avoid HoL blocking.~~
Solution: Use ghOSt to give higher priority to GET threads.

Threads notify the ghOSt scheduler about what type of request they handle.

Thread Scheduling – 50% GET – 50 % SCAN



GET Latency



SCAN Latency

Scheduling Across Layers

RocksDB workload 50% GETs -- 50% SCANs.

Problem: Most of the load comes from SCANs.

~~Solution: Add more threads to avoid HoL blocking.~~

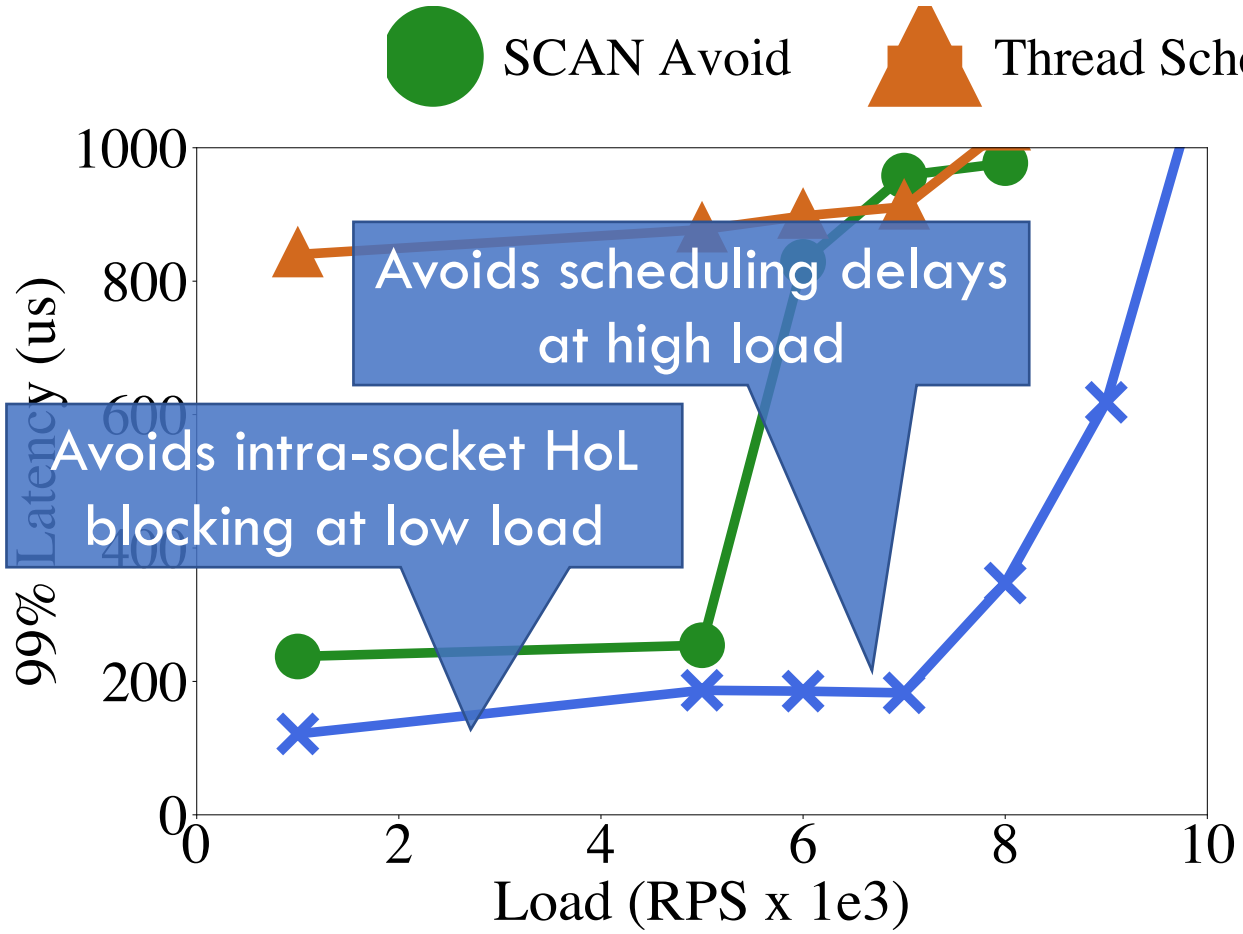
~~Solution: Use ghOSt to give higher priority to GET threads.~~

Solution: Combine SCAN Avoid + thread scheduling.

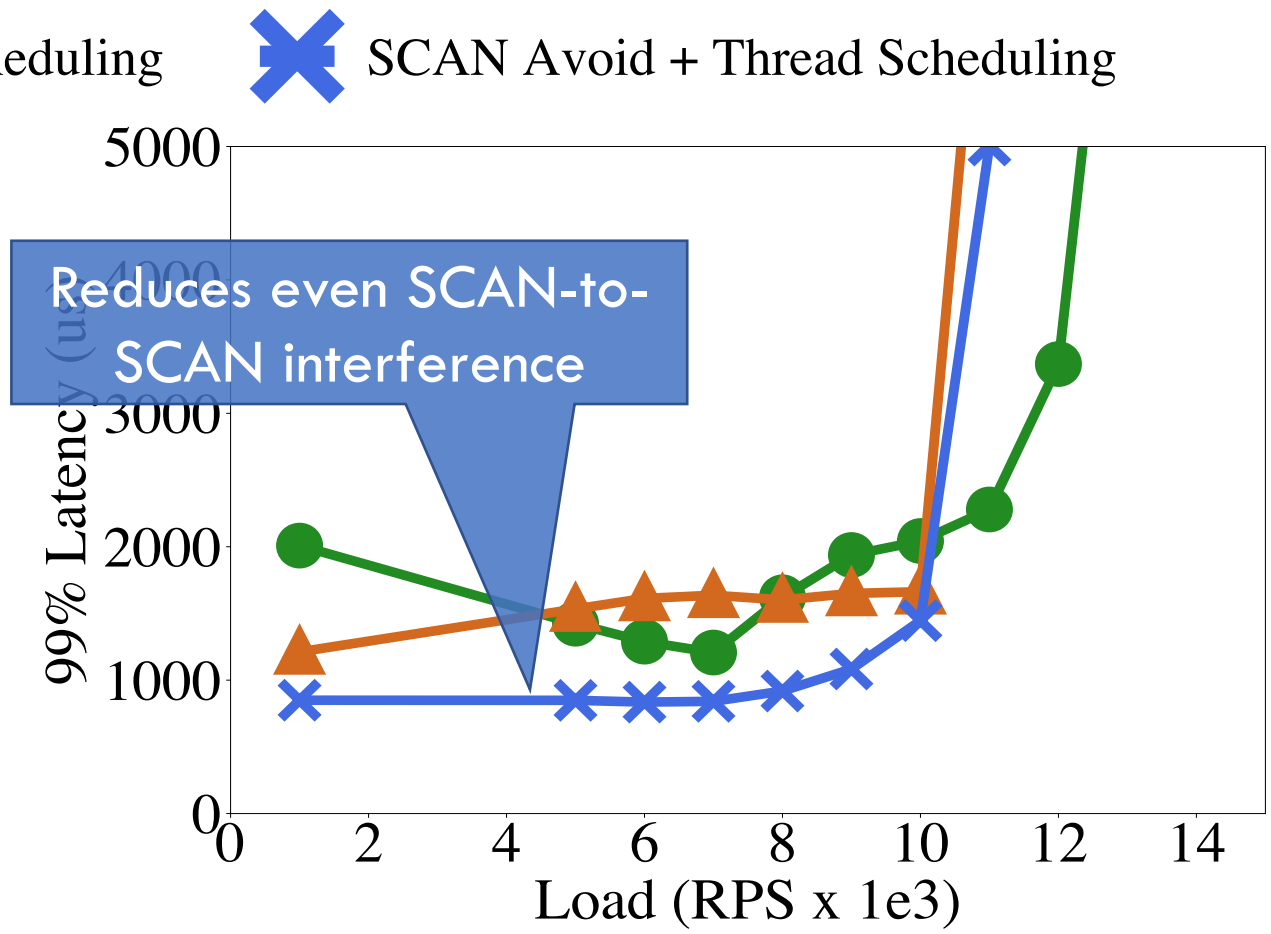
→ SCAN Avoid avoids head-of-line blocking and notifies ghOSt of the request type handled by a thread before it wakes up.

→ ghOSt thread scheduling makes sure that threads handling GETs execute immediately.

Request + Thread Scheduling – 50% GET – 50 % SCAN

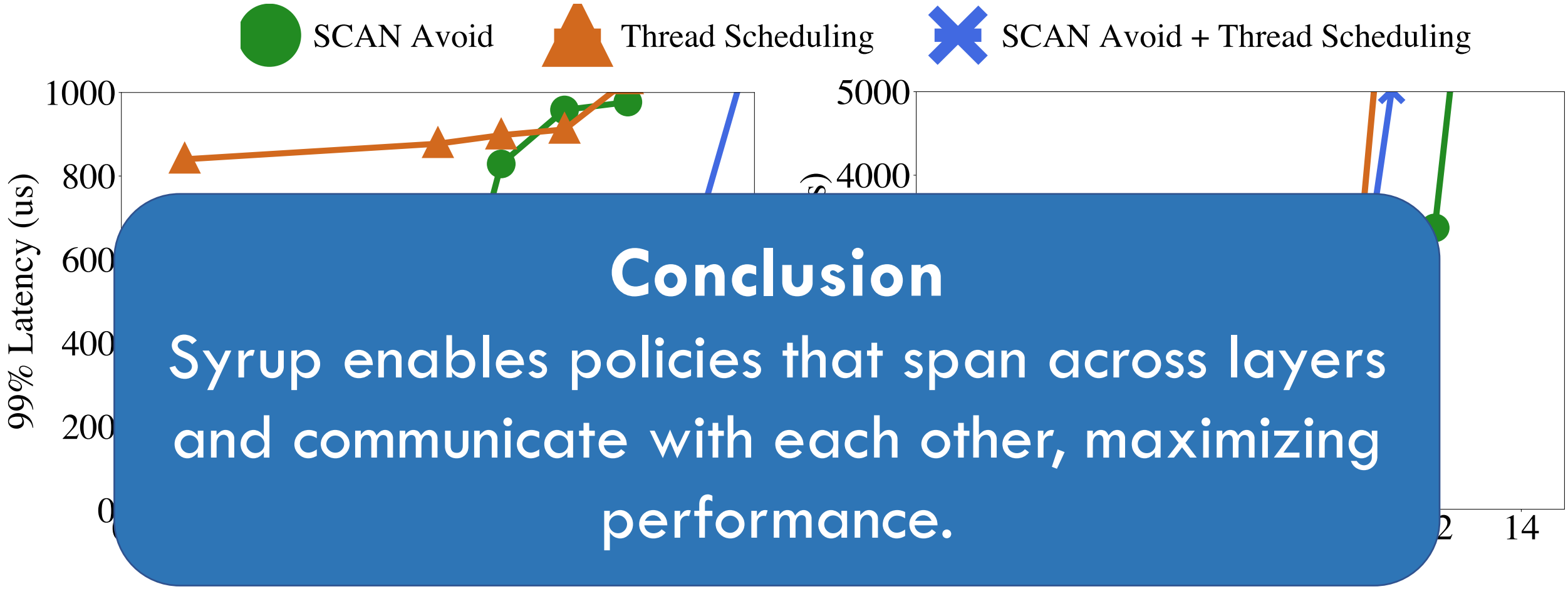


GET Latency



SCAN Latency

Request + Thread Scheduling – 50% GET – 50 % SCAN



Conclusion

Syrup enables policies that span across layers and communicate with each other, maximizing performance.

GET Latency

SCAN Latency

Syrup's Overheads

1. Policy Overhead:

Policy	LoC	Instructions	Cycles (± stdev)
Round Robin	6	56	1563 (± 89)
SCAN Avoid	21	311	1709 (± 115)
SITA	16	81	1699 (± 210)

2. Communication Overhead

→ *mmapped eBPF maps access ~= memory access*

Outline

- Motivation
- Requirements
- Syrup Design
- Evaluation
- Discussion

Scheduling over a TCP Stream

Syrup currently supports scheduling UDP datagrams and TCP connections.

→ What about intra-connection HoL blocking??

Opportunity: Add eBPF programmability to KCM (kernel connection multiplexor) and support it in Syrup.

Future Syrup Targets

The matching abstraction for scheduling is powerful and applies to most settings:

SmartNICs – Selecting an RX queue

→ Some support eBPF (Netronome example in the paper)

Switches – Selecting a port

→ Can we develop a P4 backend for Syrup?

→ How would maps work in a distributed setting?

Load Balancers – Selecting an IP address

→ Run eBPF bytecode safely in userspace?

Simplifying Programming for Syrup (and eBPF)

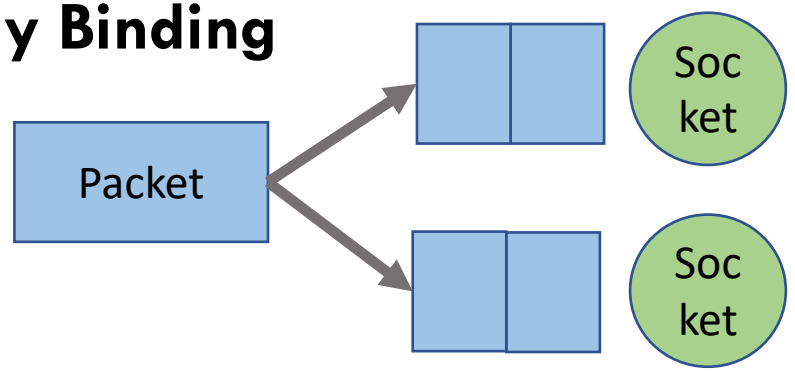
Syrup makes the declaration and deployment of scheduling policies simpler.

Users still need to write most of their policies in eBPF-compliant C code.

Question: Can we further reduce their burden?
→ Automate the bound checks.

Support for Late Binding

Early Binding

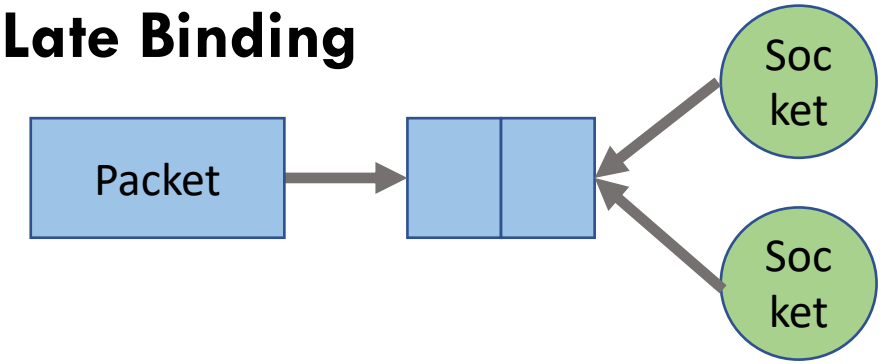


Packets are assigned to sockets upon arrival:

(+) Low-overhead

(-) HoL blocking

Late Binding



Sockets pull packets when available:

(+) No HoL blocking

(-) Higher minimum latency

Syrup in a Multi-Tenant Environment

Syrupd allows different applications to co-locate their policies.

Questions

→ What about malicious users?

→ Is their existence within an OS a realistic scenario?

→ Can we use Syrupd to safely enable non-root users to deploy eBPF programs?

Conclusion

Scheduling is a fundamental operation that:

- Varies across applications.
- Spans across different layers of the stack.
- Requires low overhead.

Syrup enables users to customize scheduling by:

- Treating scheduling as an online matching problem.
- Leveraging eBPF and ghOSt to safely and efficiently deploy scheduling policies across the stack.

Soon available at:

github.com/stanford-mast/syrup