Syrup: User-Defined Scheduling Across the Stack

Kostis Kaffes, Jack Tigar Humphries, David Mazieres, Christos Kozyrakis
Scheduling matters

• Fundamental operation in computer systems and networks.
• “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: Socket Selection

Network Sockets
listening on the same port

Hash 5-tuple
Example: Socket Selection

- **Vanilla Linux** assigns packets from 50 clients to 6 sockets using the 5-tuple hash.
Problem: Load Imbalance

Network Sockets
listening on the same port
**Solution:** Change scheduling to **Round Robin**

Network Sockets listening on the same port
Simply changing socket matching policy provides > 75% performance improvement.
Solution: Change scheduling to **Round Robin**

**Goal**
Allow developers *easily* define and deploy custom scheduling policies.

Performance improvement.
“Legacy” Scheduling Options are Hard

Implement your favorite policy in the Linux kernel:
+ Compatibility.
- Probably only a few hundred people in the world can do it.
- Hard to coordinate across layers.

Build a data-plane OS:
+ High performance / low overhead.
- Incompatible with existing applications/APIs.
- Hard and costly to maintain
Syrup makes custom scheduling easy

Policy file: Example Round-Robin Policy

```
uint32_t idx = 0;
uint32_t schedule(void *pkt) {
    idx++;
    return idx % NUM_SOCKETS;
}
```

- How to define the policy?
- How to deploy it across the stack?
  - Support for multiple apps?
  - Scheduling state management?
API for Policy Definition

How can users easily specify a scheduling policy?
The API Should Provide Expressibility

Different applications and workloads perform best under different scheduling policies:
• Low Variability $\rightarrow$ FCFS scheduling \([IX_{OSDI\ 2014},\ ZygOS_{SOSP\ 19}]\)
• High Variability $\rightarrow$ Preemption \([Shinjuku_{NSDI\ 19}]\)
  $\rightarrow$ Resource Partitioning \([Minos_{NSDI\ 19},\ DARC_{SOSP\ 21}]\)
• Memory Intensive $\rightarrow$ Locality \([MICA_{NSDI\ 14}]\)
• …
• Application-specific optimizations and request parsing
  \([MICA,\ Minos,\ Shinjuku,\ DARC]\)
Insight: Scheduling as Online Matching
Insight: Scheduling as Online Matching

- Almost declarative scheduling.
- Scheduling code portable across different layers of the stack.
- Scheduling broken down into a series of "small" decisions.
Example: Round Robin

1 uint32_t idx = 0;
2 uint32_t schedule(void *pkt) {
3     idx++;
4     return idx % NUM_SOCKETS;
5 }
6
Cross-Layer Deployment
Program the network stack with eBPF

In-kernel virtual machine that allows running sanitized user-provided code in the kernel.

Originally used for packet filtering but we realized its potential for scheduling.

LLVM compiles scheduling functions implemented in C to eBPF bytecode.
Program the network stack with **eBPF**

- **Syrup Hook**
- **User-space component**
- **Kernel component**
- **Hardware**

Network Stack Syrup Hooks

- Application Socket
  - Socket Select
    - Protocol Stack
      - XDP SKB
        - SKB Allocation
          - XDP DRV
            - NIC Driver
              - SoftIRQ Core
                - CPU Redirect
                  - RX Core
                    - XDP Offload
                      - Network Interface Card
Cross-Layer Deployment
Program kernel scheduling with ghOS\textit{t}

ghOS\textit{t} offloads kernel thread scheduling to userspace agents.

source: Humphries et al. SOSP 2021
Program kernel scheduling with ghOSt

Thread Scheduling Syrup Hook

- Syrup Hook
- User-space component
- Kernel component
- Hardware
Cross-Layer Deployment

[Diagram showing a flow from eBPF ghOSt to various scheduler components in different layers: Application, Kernel Scheduler, Networking Stack, and Network Interface Card, with checkmarks indicating successful deployment.]
Key Property: eBPF and ghOS1t properties guarantee safety

The eBPF verifier makes sure that an application policy does not “corrupt” the underlying system.

ghOS1t scheduling policies run at lower priority than CFS, allowing the system to reclaim resources.
Safe Cross-Layer Deployment
Scheduling State & Communication

STATE

- Application
- Kernel Scheduler
  - Scheduler
- Networking Stack
  - Scheduler
- Network Interface Card
  - Scheduler
Scheduling State & Communication: **eBPF maps**

A key-value store abstraction that is accessible from both user and kernel space.

*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.
Deploying Multiple Applications and Policies

![Diagram: eBPF and ghOSt with multiple applications and policies](image)
Implement a global arbiter daemon — **Syrupd** — that orchestrates the scheduling policies for different 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.

**Syrupd Provides Isolation Among Policies**
Syrup makes custom scheduling easy and safe

Policy file: Example Round-Robin Policy

```c
uint32_t idx = 0;
uint32_t schedule(void *pkt) {
    idx++;
    return idx % NUM_SOCKETS;
}
```

- Policies as matching functions
- Cross-layer deployment using eBPF and ghost
- State management and communication using maps
- Isolation between policies and the system
Specify the scheduling policy in C.
1. Specify the scheduling policy in C.

2. Deploy the policy from the application code.

---

**Application Code**

```c
... 
syr_deploy_policy(<policy_file>, <hook>);
... 
```

**Policy file: Example Hash-based Policy**

```c
uint32_t schedule(void *pkt) {
    uint32_t hash =
        hash((struct *udphdr) pkt_start);
    int num_cores =
        syr_map_lookup(core_map, 0);
    return hash % num_cores;
}
```
1. Specify the scheduling policy in C.

2. Deploy the policy from the application code.

3. Syrupd compiles the policy for the target hook(s).
1. Specify the scheduling policy in C.

```c
uint32_t schedule(void *pkt) {
    uint32_t hash = hash((struct *udphdr) pkt_start);
    int num_cores = syr_map_lookup(core_map, 0);
    return hash % num_cores;
}
```

2. Deploy the policy from the application code.

3. Syrupd compiles the policy for the target hook(s).

4. Syrupd deploys the policy to the target hook(s).
Specify the scheduling policy in C.

Deploy the policy from the application code.

Syrupd compiles the policy for the target hook(s).

Syrupd deploys the policy to the target hook(s).

The different layers can communicate using eBPF maps.
Evaluation Questions

1. Can Syrup be used to express and implement a variety of scheduling policies?
2. What are Syrup’s overheads?
3. Can Syrup be used for cross-layer scheduling?
4. Are policies implemented using Syrup portable across layers?
Syrup Usecase

We use Syrup to implement scheduling policies that match packets to network sockets.

*Before Syrup, implementing each policy required a new paper.*

- Multi-threaded **RocksDB** UDP server
- Each thread pinned to a different core
- Serving mix of 99.5% **GETs** (10 usec) and 0.5% **SCANs** (700 us)
Hooks Used: Socket Selection

Hash 5-tuple

Network Sockets listening on the same port
Vanilla Linux Policy

Very high latency even at low load due to HoL blocking

High variability at high load due to load imbalance across sockets
Round-Robin Policy

There is still HoL blocking!

Round-robin provides good load balancing
SCAN Avoid Policy

Notify kernel when handling a SCAN.

Notify kernel when finishing handling a SCAN.

```c
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);
```
SCAN Avoid Policy

Map lookup to find a non-SCAN socket.

When one is found, return the corresponding index.

```c
uint32_t schedule(void *pkt_start,
                      void *pkt_end) {
    uint32_t cur_idx = 0;
    for (int i = 0; i < NUM_SOCKETS; i++) {
        cur_idx = get_random() % NUM_SOCKETS;
        uint64_t *scan = map_lookup(&scan_map, &cur_idx);
        if (!scan)
            return PASS;
        // Stop searching when a non-SCAN socket is found.
        if (*scan == GET)
            break;
    }
    return cur_idx;
}
```
SCAN Avoid Policy

Avoids HoL blocking at low load!

There is still HoL blocking at medium and high loads.

Better

Load (RPS)

Vanilla Linux
Round Robin (6 LOC)
SCAN Avoid (21 LOC)

8x
Size Interval Task Assignment Policy

Implemented in specialized data OS [NSDI 19]!!!
Size Interval Task Assignment Policy

KERNEL

Parse the request type in the kernel.

Steer all SCANs to a specific socket.

```c
uint32_t idx = 0;

uint32_t schedule(void *pkt_start, void *pkt_end) {
    if (pkt_end - pkt_start < 16) {
        return PASS;
    }

    // First 8 bytes are UDP header.
    uint64_t *type = (uint64_t *)(pkt + 8);

    if (*type == SCAN) {
        return 0;
    }

    idx++;
    return (idx % (NUM_SOCKETS - 1)) + 1;
}
```
SITA Policy

Avoids HoL blocking even at high load!
Conclusion

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

1. Can Syrup be used to express and implement a variety of scheduling policies?
2. What are Syrup’s overheads?
3. Can Syrup be used for cross-layer scheduling?
4. Are policies implemented using Syrup portable across layers?
Syrup’s Overheads

1. Policy Overhead:

<table>
<thead>
<tr>
<th>Policy</th>
<th>LoC</th>
<th>Instructions</th>
<th>Cycles (± stdev)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round Robin</td>
<td>6</td>
<td>56</td>
<td>1563 (± 89)</td>
</tr>
<tr>
<td>SCAN Avoid</td>
<td>21</td>
<td>311</td>
<td>1709 (± 115)</td>
</tr>
<tr>
<td>SITA</td>
<td>16</td>
<td>81</td>
<td>1699 (± 210)</td>
</tr>
</tbody>
</table>

2. Communication Overhead

→ *mmapped eBPF maps access ~ = memory access*
Evaluation Questions

1. Can Syrup be used to express and implement a variety of scheduling policies?
2. What are Syrup’s overheads?
3. Can Syrup be used for cross-layer scheduling?
4. Are policies implemented using Syrup portable across layers?
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 sockets to avoid HoL blocking.
SCAN Avoid – 50% GET – 50% SCAN

GET Latency increases as all cores are occupied by SCANS.

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 ghOSSt to give higher priority to GET threads.

Threads notify the ghOSt scheduler about what type of request they handle.
GET Latency is high as there is still intra-socket HoL blocking.
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 ghOSSt to give higher priority to GET threads.
Solution: Combine SCAN Avoid + thread scheduling.

→ SCAN Avoid avoids head-of-line blocking and notifies ghOSSt of the request type handled by a thread before it wakes up.
→ ghOSSt thread scheduling makes sure that threads handling GETs execute immediately.
Request + Thread Scheduling – 50% GET – 50% SCAN

- Avoids intra-socket HoL blocking at low load
- Avoids scheduling delays at high load
- Reduces even SCAN-to-SCAN interference
Hooks Used: **Socket Selection & Thread Scheduler**

- Syrup Hook
- User-space component
- Kernel component
- Hardware

**Thread Scheduling Syrup Hook**
- Application Thread
- Application Thread
- ghOSt
- Thread Scheduler
- Core
- Core

**Network Stack Syrup Hooks**
- Application Socket
  - Socket Select
  - Protocol Stack
  - XDP SKB
  - SKB Allocation
  - XDP DRV
  - NIC Driver
  - SoftIRQ Core
  - CPU Redirect
  - RX Core
  - XDP Offload
- Network Interface Card
Conclusion

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

1. Can Syrup be used to express and implement a variety of scheduling policies?
2. What are Syrup’s overheads?
3. Can Syrup be used for cross-layer scheduling?
4. Are policies implemented using Syrup portable across layers?
Policy Portability

MICA Key-Value Store [NSDI’14]
Avoids locking by partitioning the key space to hardware threads. Performance limited by data movement.

![Diagram showing partitioning of key space between Application HTs and Networking HTs]
Vanilla MICA Performance

99.9% Latency (us)

Load (RPS x 1e6)
Policy Portability – Syrup SW

We use Syrup to choose the correct socket based on the key and eliminate the app-to-app communication.
Syrup Code

```c
uint32_t schedule(void *pkt_start,
                   void *pkt_end) {
    uint32_t key = get_key(pkt_start,
                            pkt_end);
    return key % NUM_EXECUTORS;
}
```
Syrup deployed in SW eliminates a data movement improving performance.
Policy Portability -- NIC layer

We use the same policy on the NIC to steer each packet to the hyperthread “buddy” of its “home” core.
Hooks Used: **NIC - XDP Offload**

- Syrup Hook
- User-space component
- Kernel component
- Hardware

**Thread Scheduling Syrup Hook**

- Application Thread
- Application Thread
- ghOSSt
- Thread Scheduler
- Core
- Core

**Network Stack Syrup Hooks**

- Application Socket
- Application Socket
- Protocol Stack
- Protocol Stack
- XDP SKB
- XDP SKB
- SKB Allocation
- SKB Allocation
- XDP DRV
- XDP DRV
- NIC Driver
- NIC Driver
- SoftIRQ Core
- SoftIRQ Core
- CPU Redirect
- CPU Redirect
- RX Core
- RX Core
- XDP Offload
- Network Interface Card

Hooks Used: NIC - XDP Offload
Syrup HW MICA Performance

Syrup deployed in HW eliminates all endhost data movement.

- Vanilla MICA
- Syrup SW (AF_XDP)
- Syrup HW (NIC)
Conclusion

Syrup policies are portable to different hooks depending on the capabilities of the underlying system and hardware.
Conclusion

Scheduling is a fundamental operation that:
• Varies across applications.
• Spans across different layers of the stack.
• Requires low overhead.

Syrup enables users to **easily** 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](https://github.com/stanford-mast/syrup)