Syrup: User-Defined Scheduling Across the Stack

Kostis Kaffes
Stanford University

Jack Tigar Humphries
Stanford University

David Mazières
Stanford University

Christos Kozyrakis
Stanford University

Abstract

Suboptimal scheduling decisions in operating systems, networking stacks, and application runtimes are often responsible for poor application performance, including higher latency and lower throughput. These poor decisions stem from a lack of insight into the applications and requests the scheduler is handling and a lack of coherence and coordination between the various layers of the stack, including NICs, kernels, and applications.

We propose *Syrup*, a framework for user-defined scheduling. Syrup enables *untrusted* application developers to express application-specific scheduling policies across these system layers without being burdened with the low-level system mechanisms that implement them. Application developers write a scheduling policy with Syrup as a set of matching functions between inputs (threads, network packets, network connections) and executors (cores, network sockets, NIC queues) and then deploy it across system layers without modifying their code. Syrup supports multi-tenancy as multiple co-located applications can each safely and securely specify a custom policy. We present several examples of uses of Syrup to define application and workload-specific scheduling policies in a few lines of code, deploy them across the stack, and improve performance up to 8× compared with default policies.

CCS Concepts: • **Networks** → *Programmable networks*; • **Software and its engineering** → **Scheduling**.

Keywords: scheduling, programmability, kernel

ACM Reference Format:
Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-Defined Scheduling Across the Stack. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483548

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSP 2021, October 25–28, 2021, Virtual Event, Germany
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483548

1 Introduction

Scheduling is a fundamental operation in computer systems that occurs at multiple layers across the stack, from global load balancers to programmable network devices to the operating system in end-hosts. Despite this diversity, schedulers perform a single fundamental operation: they map work to execution resources. The units of work range from network packets to OS threads to application-level requests, while execution resources include NIC queues, cores, and network sockets, depending on where scheduling occurs.

The scheduling policy used for each application is critical to its performance; different applications and workloads perform best under different policies. Matching scheduling policies to the application characteristics can reduce or eliminate problems such as head-of-line blocking [27, 38], lack of work conservation [32, 42], priority inversion [45], and lack of scheduler scalability [15, 31, 43]. Given the wide range of algorithms, tuning parameters, and implementation options, the performance difference from using a generic scheduling policy versus an application-specific scheduling policy is often an order of magnitude or more [18, 27, 38, 40, 42].

Unfortunately, the scheduling policy is typically baked into the design of most systems. Applications are forced to use the default policies across system layers which are often suboptimal. For example, the complex Linux scheduler and its Completely Fair Scheduler (CFS) policy were found to be non-work-conserving for important application classes [35]. At the same time, widely-used hash-based packet steering (RSS) is known to lead to load imbalances affecting application tail latency [13, 27, 43]. Application developers can hack existing full-featured operating system components — from scheduling classes to the networking stack to NIC and flash card drivers — in order to implement custom policies. However, this is extremely hard to do in the first place and equally hard to upstream and maintain in the long term. One should not need to be a Linux kernel contributor to optimize their application. Hence, developers who have wanted to use custom scheduling even for existing applications have built runtimes, operating systems, and even hardware from scratch [14–16, 27, 31, 37, 38, 42, 51], sacrificing compatibility with existing APIs, applications, and hardware. For example, even a popular data plane operating system like IX [14] only

supports three NICs belonging to the same vendor (Intel), while recent designs propose keeping the existing kernel API and using specialized hardware to accelerate the data plane [44].

We argue that rather than resort to kernel hacking or building specialized full-stack systems that optimize scheduling for each new class of workloads, *application developers should be able to express their preferred scheduling policies to the underlying systems.* The high-level policy code should be automatically integrated with the various scheduling mechanisms throughout a modern system, delivering a significant fraction of the performance benefits possible with an application-specific operating or runtime system.

To address this need, we developed *Syrup, a framework for user-defined scheduling*. As shown in Figure 1, untrusted application developers can use Syrup to define and safely deploy arbitrary scheduling policies across various layers of the stack, including the kernel networking stack, the kernel scheduler, and programmable network devices. For example, developers can specify how threads are scheduled to cores or how packets are scheduled to network sockets.

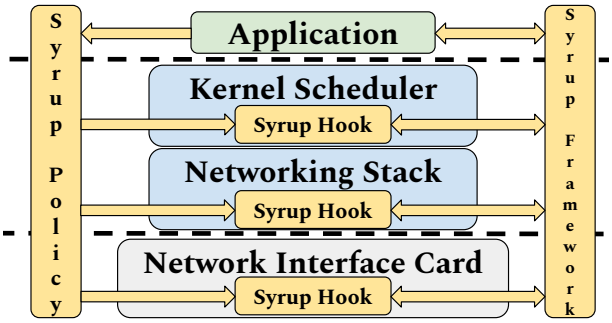


Figure 1. Syrup enables user-defined scheduling across the stack.

Syrup makes specifying scheduling policies easy by treating scheduling as a matching problem. Users specify scheduling policies by implementing functions that match inputs to executors without dealing with low-level system details such as how inputs are actually steered to the selected executors. Syrup currently supports a range of inputs including network packets, network connections, and kernel threads, while executors can be NIC queues, cores, and network sockets. Developers just need to declare their scheduling intention and Syrup does everything else, *making scheduling almost declarative.* Syrup introduces system hooks so that policies are deployed efficiently across different layers of widely-used software and hardware stacks. Syrup also allows communication between scheduling policies that run in different system layers using a `Map` abstraction, and guarantees inter-application isolation, ensuring that policies only handle inputs belonging to the app that deployed them.

Syrup currently provides hooks for three backends: eBPF [3] software, eBPF hardware, and ghOSt [25]. eBPF software is used to safely deploy matching functions at various hooks in the kernel networking stack, while eBPF hardware allows Syrup to take advantage of programmable network devices. ghOSt [25] is used to offload thread scheduling to matching functions that run in userspace.

We show Syrup can concisely express and efficiently implement the workload-specific scheduling policies needed for demanding workloads. First, we demonstrate that several scheduling policies with different requirements can be implemented in a few lines of code and deployed for socket selection for a multi-threaded RocksDB [2] workload in Linux while achieving up to 8× lower tail latency and 3× higher throughput compared to the default policy. The requirements range from peeking into packet contents to identify the request type to communicating with a userspace agent that generates tokens consumed by different users in an SLO-aware policy. Using Syrup, we also deploy custom policies that work in concert across the network stack and the thread scheduler, improving the performance of a different RocksDB workload by 60% compared to single-layer scheduling. Finally, we show that the same Syrup-based scheduling policy is portable across different layers of the stack by moving scheduling for MICA [34], a high-performance key-value store that serves millions of requests per second, between a smartNIC and the network stack.

The rest of the paper is organized as follows. §2 motivates user-defined scheduling across the stack. §3 presents the design of Syrup and §4 describes its implementation. §5 evaluates how applications benefit from Syrup-based scheduling. Finally, §6 discusses open research questions and potential Syrup extensions, while §7 presents related work.

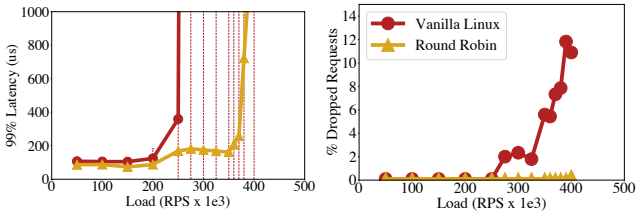
2 Motivation

2.1 User-defined Scheduling Matters

To showcase the importance of scheduling, we examine a very simple workload scenario. We set up a 6-thread RocksDB server [2] that handles homogeneous GET requests with a service time of 10-12μs. All threads have a socket bound to the same UDP port and we let Linux distribute incoming datagrams to sockets. More details about the experimental setup can be found in §5.2.

This very common case should be a slam-dunk for the vanilla scheduling policy in Linux that assigns datagrams to sockets using the hash of the 5-tuple of each datagram. However, in Figure 2, we see that this policy leads to many dropped requests and high and noisy 99% latency for high request rates (> 250K RPS). Most of the data points for vanilla Linux are so high that all we see is the lower part of the standard deviation bars across twenty runs. Such behavior has been observed before [13, 27] and is happening because the hash-based scheduling scheme can cause imbalances when there is a small number of 5-tuples (50) and sockets

(6), and more 5-tuples than expected are randomly assigned to the same socket, overloading it. This is a major problem as most cloud SLOs are set in terms of tail latency. A simple round-robin policy over network sockets implemented in Syrup eliminates drops and achieves sub-200μs tail latency for a load 80% higher than the default policy.



(a) 99% Latency **(b)** % Dropped Requests

Figure 2. RocksDB benchmark with 100% GET requests.

It is important to note that while the round-robin policy is better for this workload, it is no panacea. For other workload types, locality might matter more. Optimizations like Linux's Receive Flow Steering (RFS) that places network processing on the same core as the receiving application would be impossible without hash-based scheduling. A netperf TCP_RR test that uses RFS has been shown to achieve up to 200% higher throughput than one without RFS [1]. Hence, scheduling flexibility and customizability is a necessary feature for modern operating systems.

Most existing systems do not offer scheduling flexibility. The upstream Linux kernel essentially supports just six scheduling policies (CFS, BATCH, IDLE, FIFO, RR, DEADLINE), and adding a new policy even to a custom kernel requires significant development effort [37]. Hence, application developers often build a new bespoke framework or data plane system for each application class they want to schedule differently [14, 16, 24, 27, 31, 34, 38, 40, 42, 51]. This approach has significant shortcomings:

- Considerable development effort is necessary even to prototype new scheduling policies.
- These dataplanes typically use special APIs that are incompatible with commonly-used applications built on top of existing systems, e.g., Linux.
- Maintaining such specialized per-application systems as the infrastructure around them changes is time-consuming and costly.

Instead of building new runtimes or operating systems for each application class, we argue that application developers should specify their preferred scheduling policy and safely deploy it to existing systems. Schedulers throughout the system stack should take the application preferences into account to optimize for workload-specific patterns.

2.2 Scheduling Requirements

In this section, we examine an important application class, key-value stores (KVS), and derive a minimum set of requirements that a user-defined scheduling framework must satisfy. Key-value stores are widely used in web applications, and their diverse workloads and high performance requirements present a wide variety of scheduling challenges.

Expressibility: Different workloads perform best under different scheduling policies, even within a single application type. For example, homogeneous workloads where requests have about the same execution time are better served by low-overhead, First-Come-First-Serve (FCFS) policies [14, 28, 34]. For workloads with higher execution time variability, FCFS often leads to head-of-line blocking. Policies that use work-stealing [42] or centralized scheduling and preemption [27] are better suited for such workloads, achieving up to 8× better performance. In more extreme bimodal scenarios, it might even be necessary to reserve some cores for each request type using the Size Interval Task Assignment policy [20] or one of its variations [16, 38]. Therefore, a scheduling framework must allow applications to define custom policies easily.

Cross-layer deployment: Scheduling is not limited to a single layer of the stack. Prior work has shown that implementing a policy at the right layer can greatly improve performance. In some cases, it is beneficial to offload packet scheduling to hardware to improve scalability [14, 28, 31]. However, in other cases, more complex policies running in software can lead to better performance [27]. More importantly, some systems improve performance even further by implementing scheduling policies that work in concert across multiple layers of the stack [26, 29, 51]. For example, locality constraints that are important for high-performance networked applications [14, 34] need to be enforced both at the NIC RX queue and at the application thread layers.

Low Overhead: Many modern cloud workloads, including but not limited to key-value stores, in-memory databases, web search, and interactive analytics, operate on the microsecond scale. Hence, scheduling mechanisms and policies that operate per packet or I/O event should incur very little overhead. Applications with end-to-end latencies on the order of a few tens of microseconds cannot tolerate scheduling delays higher than single-digit microseconds.

Multi-tenancy and Isolation: One of the main disadvantages of most of the aforementioned custom data plane and runtime systems is that hosting more than one application requires starting multiple dedicated system instances. Even systems built for multiplexing, e.g., Shenango [40], offer no scheduling flexibility, using the same policies for all application and workload mixes. This can severely limit application performance as there is no one-size-fits-all scheduling policy. Moreover, any system with built-in support for flexibility will need safeguards to ensure that different scheduling policies, one for each application, can safely co-exist.

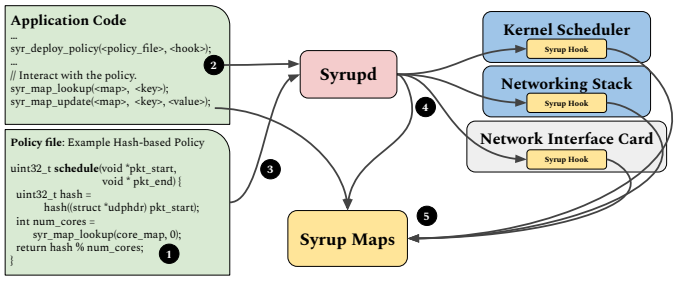


Figure 3. Scheduling workflow in Syrup.

3 Syrup Design

Syrup is a framework that enables user-defined scheduling while meeting all of the requirements mentioned above. Application developers write a custom scheduling policy for their application using Syrup and then deploy the untrusted code safely and efficiently across system layers in the data center. Scheduling policies expressed in Syrup can move and split across layers as needed with minimal effort.

3.1 Workflow Overview

Figure 3 summarizes the key components of Syrup and the typical scheduling workflow. First, the developer or the administrator of an application that wants to use Syrup specifies her desired scheduling policy by implementing a simple C interface in a separate file ❶. In §3.2, we describe how Syrup adopts a matching abstraction for scheduling that allows developers to specify policies in an almost-declarative fashion. The application code then calls the `syr_deploy_policy` function, which takes two arguments: a file describing the desired scheduling policy and one or more target deployment hooks for the policy ❷. This function communicates with a system-wide Syrup daemon, `syrupd`, which does all the heavy lifting for the policy deployment. The daemon compiles the policy file to a binary or object file ❸ — depending on the target scheduling hook — and deploys it in the user-specified hooks across the stack ❹. The userspace application and the Syrup policies deployed across different hooks can optionally communicate information such as load, latency statistics, or expected completion time using a key-value store-like `Map` abstraction ❺. `Maps` are defined in the policy file and set up by `syrupd` at deployment time. Applications can update or deploy new policies at any time while they are running. If no Syrup policy is deployed, the application runs using the default scheduling policy of the underlying runtime and operating system.

3.2 Scheduling as a Matching Problem

Syrup aims to offer a single scheduling abstraction that can be used for different scheduling decisions across the stack and make it easy for users to implement custom scheduling policies. Syrup achieves this by treating *scheduling as fundamentally an online matching problem.* Scheduling policies are represented as matching functions between *inputs and executors that process the inputs.* Inputs can be any supported units of work and executors can be any system processing components. Syrup currently supports network packets, connections, and threads as inputs and NIC queues, cores, and network sockets as executors. Syrup policies run whenever a new input is ready for processing, e.g., a packet arrives or a thread becomes runnable, or an executor becomes available. Implementing Syrup support for additional inputs (I/O operations) and executors (NVMe queues) that cover storage use cases is straightforward [49].

Syrup's matching abstraction offers generality as it can be used for decisions as fine-grained as placing packets to cores for network stack processing to as large-scale as placing jobs to machines in a data center. Moreover, online matching breaks scheduling down into a series of "small" decisions, improving the composability and the understandability of even complex policies. For example, optimizing packet processing in Linux translates first to assigning packets to NIC queues, then to cores for network stack processing, and eventually to application-level sockets. A Syrup user can define and deploy a different self-contained scheduling policy for each of these decisions.

Implementing scheduling as per-application online matching also offers reliability and isolation advantages. Syrup makes sure that each policy only processes inputs belonging to the application that deployed it. A bad-performing or buggy policy will only affect the application that deployed it, leading to improved reliability over monolithic system-wide policies. Malicious applications can only affect overall system performance by hogging some executors, a behavior quickly detected and dealt with using a resource manager. We discuss in detail how Syrup meets these isolation guarantees in §3.5 and §4.3.

3.3 Specifying a policy in Syrup

To define a scheduling policy, users simply need to provide an implementation of the `schedule` matching function (see Table 1) that is then deployed to a scheduling hook by `syrupd`. The only thing this user-defined function needs to do is select an executor for the input passed to it as an argument. The actual enforcement of the scheduling decision, e.g., assigning a packet to a specific network socket when SO_REUSEPORT is used, is hook-specific and handled exclusively by the Syrup framework. This almost-declarative API removes most of the programming burden from the policy developer.

Syrup's `schedule` function is expected to return a `uint32_t` instead of an actual executor object. The return value is a key to an application- and hook-specific `Map` set up by `syrupd` at deployment time. This `Map` holds the set of available executors, e.g., in the case of connection scheduling, the corresponding `Map` stores network sockets. There are also two special return values, PASS and DROP, that signal to the system to use its default policy or drop the input respectively. Syrup users can populate this `Map` with executors as they see fit, e.g., add network sockets after `bind()` is called.

Syrup API			
Function Name	**Arguments**	**Output**	**Description**
schedule	input	executor	Implements the scheduling policy by matching the <input> with an <executor>; written in a safe subset of C
syr_deploy_policy	policy_file, hook	prog_fd	Deploys the policy in <policy_file> to scheduling <hook>
syr_map_open	path	map_fd	Opens the Map pinned to <path>
syr_map_close	map_fd	status	Closes Map associated with <map_fd>
syr_map_lookup_elem	map_fd, key	value	Returns the <value> associated with <key> in <map_fd>
syr_map_update_elem	map_fd, key, value	status	Stores <value> to <map_fd>'s <key>

Table 1. The Syrup API. `schedule` is implemented by Syrup users while the rest of the functions are provided by the framework.

This design choice makes Syrup policies more portable as they can be reused without any code change across layers that support the same inputs. For example, the following hash-based policy can assign UDP packets to NIC queues, cores, or application-level sockets. In §5.4, we show that this simple policy implemented in Syrup improves throughput by more than 80% for a high-performance key-value store application.

```
uint32_t schedule(void *pkt_start, void *pkt_end) {
  uint32_t hash = hash((struct *udphdr) pkt_start);
  return hash % NUM_EXECUTORS;
}
```

Similarly, porting a policy to a hook that handles different inputs typically requires minimal changes in the input-handling code, e.g., hashing the TCP header instead of the UDP one. §4.3 explains why we represent a packet using two pointers, one pointing to the start of the packet and one to the end. We present more policy examples in §5.2.

3.4 Cross-layer communication

§2.2 motivated that the runtime communication between the application code and the schedulers deployed across different layers of the stack is often crucial for scheduling performance and functionality. For example, resource allocation decisions using expensive learning-based techniques are often offloaded to userspace, outside of the critical path [36, 47].

In Syrup, such communication is done using Maps that provide a key-value store API, as shown in Table 1. In addition to the Maps used as executor containers, applications can declare and populate custom Maps. User-defined Maps are declared in the policy file and pinned to `sysfs` by `syrupd` so that different programs from the same user can access them. We can control access to maps using file system permissions. Our implementation supports application-defined Maps with 32-bit unsigned integer keys and arbitrary C structs as values. However, we have found that 64-bit unsigned integer values are sufficient for our target applications and policies. Hence, to simplify the API, we use by default 64-bit values. We discuss the atomicity model of Maps in §4.1 and the different operations' overhead in §5.5.

To showcase the usefulness of Maps and the scheduling flexibility provided by Syrup, we develop a token-based resource allocation policy similar to the one used by Reflex [30]. To avoid SLO violations, this policy issues tokens to each user periodically. Incoming requests consume tokens, and if a user's tokens drop to zero, requests are dropped. We select a token generation rate so that the system operates slightly below its saturation rate.

```
1  struct app_hdr {
2    uint32_t user_id;
3    ...
4  }
5
6  uint32_t schedule(void *pkt_start, void *pkt_end) {
7    void * data = (void *) (pkt_start + sizeof(struct udphdr));
8    struct * app_hdr = (struct app_hdr *) data;
9
10   uint32_t user_id = app_hdr->user_id;
11
12   uint64_t * tokens = syr_map_lookup_elem(&token_map, &user_id);
13   if (*tokens == 0) {
14     return DROP;
15   } else {
16     __sync_fetch_and_add(tokens, -1);
17     return PASS;
18   }
19 }
```

In the code snippet above, we omit some of the necessary bound checks for brevity. The scheduling code first parses each UDP packet to identify the user the packet belongs to (line 10). It then does a Map lookup to find the number of available tokens for the user (line 12). If there are no tokens available, it signals to the system to drop the input (lines 13-14). If there are available tokens, it decreases the token number for the user and signals to the system to choose its default executor (lines 15-17). Syrup code that runs in the kernel can directly update the value of a Map using atomic instructions (line 16). Code running in userspace can periodically replenish the tokens for each user, as shown below:

```
void generate_tokens(int token_fd, uint32_t user_id, uint64_t
      tokens) {
  syr_map_update_elem(token_fd, user_id, tokens);
}
```

To conclude, Maps provide a general communication abstraction that enables cross-layer communication while catering to evolving application needs.

3.5 Syrupd for multi-tenancy and isolation

Syrupd provides cross-application isolation as motivated in §2.2. Before loading a policy into a scheduling hook, it installs checks that ensure that each policy handles only inputs belonging to the policy's application. Applications could also load their policies directly into the different Syrup hooks but, without syrupd, there would be no safeguards protecting applications from each other. We discuss how syrupd installs the input checks in §4.3.

4 Syrup Implementation

To enable user-defined scheduling, we must safely deploy user code in the kernel and hardware or offload scheduling decisions to userspace. Syrup's implementation supports both of these approaches by using two recently-developed frameworks, eBPF [3] and ghOSt [25]. We use eBPF to safely deploy policies across the Linux networking stack and programmable NICs and ghOSt to offload kernel thread scheduling to userspace agents. As we show in §5.3, Syrup policies can be deployed and interoperate using both frameworks at the same time.

4.1 eBPF & ghOSt

eBPF Extended Berkeley Packet Filter, eBPF, is an in-kernel virtual machine that allows running userspace-provided code in the kernel in a sanitized way. eBPF programs are loaded into the kernel using a system call, attached to a specific code path and triggered by events, e.g., the arrival of a network packet. The eBPF program is loaded in bytecode that is recognized by the virtual machine and is compiled at load time to the underlying hardware instruction set. Just-in-time (JIT) compilation allows eBPF programs to achieve native code performance and their invocation to be as cheap as a regular function call. eBPF is supported by the LLVM/Clang toolchain allowing users to write scheduling policies for Syrup in a safe subset of C (see §4.3) and compile them to eBPF bytecode using the `"-target bpf"` flag. Implementing Syrup policies that will be deployed to an eBPF hook is as simple as writing a C function. We provide such policy examples in §5.2. Syrup handles deployment and ensures that the right inputs are handled by each application's policy.

As we already described in §3.4, Syrup uses Maps for cross-layer communication and for allowing applications to modify the possible executors that handle their inputs. Syrup Maps are implemented using eBPF maps. eBPF maps are kernel data structures that can store arbitrary values and can be pinned to sysfs so that multiple programs can have access to them. In Syrup we use eBPF maps that can either hold different executors, e.g., CPUs or network sockets, as well as simple uint64_t values. We describe in more detail how users specify executors and inputs in §4.4. One disadvantage of BPF maps is that they do not support synchronization primitives, e.g., locks. However, it is possible to use atomic instructions directly on BPF map values. The existence of this feature, together with the fact that scheduling does not – in most cases – need to be deterministic, means that the lack of synchronization primitives neither affects our ability to implement policies in Syrup in practice nor undermines system stability.

ghOSt eBPF is an excellent backend for network stack scheduling due to the security (§4.3), low overhead (§5.5), and flexibility it provides (§4.2). However, eBPF's support for kernel thread scheduling is not very mature. There are very few eBPF hooks in the kernel scheduler subsystem, and most of them are associated with observability, not decision-making. Thus, we opt to offload thread scheduling to userspace using ghOSt instead. The coarser granularity of thread scheduling allows us to do so, while, e.g., offloading per-packet scheduling decisions to userspace would add too much context-switch overhead.

ghOSt [25] is a new Linux kernel scheduler that implements centralized scheduling and has an easy-to-use API for implementing new centralized policies. ghOSt allows policies to be implemented in a single spinning *userspace* thread, similar in spirit to a microkernel or Exokernel [17], in order to reduce development and debugging overhead. Syrup users follow the same workflow as in Figure 3, i.e., they implement a scheduling policy as a C function that takes kernel threads as inputs and matches them with logical cores. Similar to userspace application code, Syrup policies deployed in ghOSt can use Maps to communicate with the application or policy code deployed to other hooks across the stack. ghOSt contains a lightweight kernel scheduling class that detects interesting scheduling events, such as thread state changes (e.g., thread created, thread blocked, thread yielded, etc.) and notifies the userspace process of these state changes via a message-passing API. The spinning userspace thread processes these messages, updates its own state, and then invokes the user-defined scheduling function to make a decision. The spinning userspace thread notifies the kernel of its scheduling decisions via a system call. The kernel acts on those decisions by sending interrupts to the remote logical cores that are being rescheduled and then context switching to the scheduled threads on those remote cores.

4.2 Supported Hooks

As shown in Figure 4, our initial implementation of Syrup supports scheduling hooks across the networking stack and the kernel scheduler. There is a Thread scheduler hook that allows Syrup users to define a scheduling policy that matches threads to cores using ghOSt. The Socket Select hook enables policies that choose one of many TCP or UDP network sockets that use the SO_REUSEPORT option to listen for incoming connections or datagrams on the same port. The CPU Redirect hook allows users to steer packets to specific cores for kernel network stack processing. The XDP_DRV and XDP_SKB hooks allow Syrup to handle packets early in

the network stack, before going through protocol processing. Syrup code running in these hooks can redirect packets directly to AF_XDP network sockets [23] in userspace achieving kernel-bypass-like performance on top of the Linux kernel. We describe these hooks and their use cases in more detail in §5.4. Finally, the XDP Offload hook allows users to run their policies on smartNICs and programmatically steer packets to the NIC RX queues.

4.3 Cross-application Isolation

Syrup allows the simultaneous deployment of multiple user-defined policies for different applications and offers the following guarantees:

- A policy loaded by one application only has access to the inputs belonging to that application.
- The policy code loaded by one application cannot make unauthorized accesses to memory belonging to other applications or the kernel.

We examine how Syrup achieves these two guarantees in each of its backends, eBPF and ghOSt.

eBPF Isolation Injecting user-defined code into the kernel is fundamentally dangerous regardless of Syrup's requirements. Hence, the eBPF framework offers an in-kernel verifier that performs a number of checks before the eBPF program is loaded into the kernel. The verifier simulates the execution of the program one instruction at a time and checks for out-of-bound jumps and out-of-range data accesses, while it allows pointer accesses only after an explicit check for bound violations. This is the reason why we need to include pointers to both the packet start and the packet end as input arguments in the policy shown in Figure 3. Every memory access to the packet needs to explicitly check whether it goes out of bounds, exceeding the `packet_end` pointer. Moreover, the verifier analyzes up to 1 million instructions. If there is a possibility that the loaded eBPF program exceeds that threshold, the verifier rejects it to guarantee liveness. As a result of this, only bounded loops are allowed, a restriction that does not affect the design and implementation of scheduling policies as we found out empirically. These eBPF features satisfy the second guarantee, i.e., they do not allow unwarranted memory accesses.

However, satisfying the first guarantee while using eBPF is not trivial. eBPF was primarily designed as a system administrator tool; its developers expected eBPF programs to be loaded by a root user and apply system-wide policies. For example, BPF programs loaded in the lower parts of the networking stack are triggered and have access to every network packet regardless of the application.

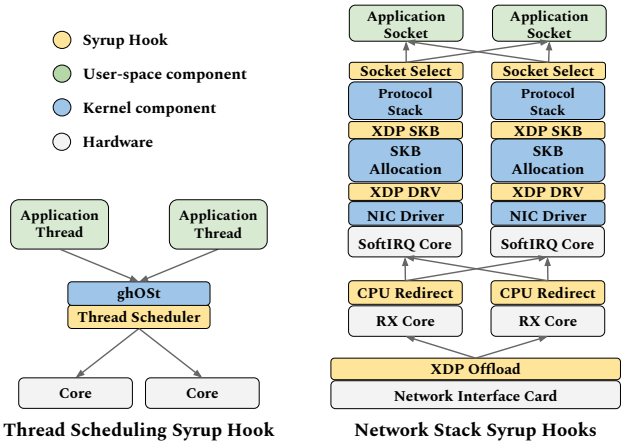
To avoid such issues, we offer Syrup-as-a-service through `syrupd`, a long-running daemon that is using a Unix Domain Socket to listen for requests from applications. An application calling the `syrup_deploy_policy()` function sends a request to that daemon instead of loading the eBPF program into the kernel itself. The request includes the policy file name and the hook. The daemon keeps track of which port belongs to which application and dynamicaly deploys the new policy making sure that each application's program handles only packets directed to its corresponding port. It maintains an eBPF map of type `PROG_ARRAY` and at start-up loads an eBPF program that matches port numbers of incoming requests to entries in the `PROG_ARRAY` map. When the daemon receives the request to load a new application-specific policy, it loads the BPF bytecode for the policy in the `PROG_ARRAY` map and inserts the corresponding port-matching rule that makes a tail call to the corresponding policy program. This design makes sure that policies of each application only handle inputs belonging to that application.

ghOSt Isolation Each application wants to implement its own Syrup policy for thread-scheduling, and importantly, each application's Syrup policy must schedule only the threads and access only the memory belonging to its own application, not those of the kernel, other policies, or other apps. ghOSt has the right isolation mechanisms in place to meet Syrup's protection guarantees. Each application's Syrup policy launches a new userspace ghOSt scheduling process. The ghOSt kernel code ensures that each Syrup thread policy running in a ghOSt userspace process can only see thread state and can only schedule threads that belong to its own application. Furthermore, policies are unable to access memory that does not belong to them because each ghOSt policy is isolated to its own address space. ghOSt policies run at a lower priority than CFS, so a malicious application cannot take over and starve the system.

4.4 Specifying Inputs and Executors

`Syrupd` and Syrup users must specify the inputs each policy handles and the executors the policy can use. Specifying executors is straightforward. For hooks in the upper layers of the network stack, the executors are networks sockets that listen for datagrams/connections on the same port. Users register the first socket they create with Syrup using the socket's and the program's file descriptors and then add this and subsequent sockets to the relevant executor map created when the policy is deployed. The application controls the map index used for each socket, allowing the policy function to make a scheduling decision simply by returning an index to that map. For the rest of the scheduling hooks, executors are hardware resources of the server, i.e., either CPU cores or NIC queues. `Syrupd` currently statically allocates a number of these resources per application and adds them to per-policy maps using the core or the queue id. Similar to the socket case, policies make a scheduling decision by returning an index to the relevant map.

In the case of inputs, there are differences between eBPF and ghOSt backends. For the upper layers of the networking stack, each Syrup program is associated with a set of sockets.



Hook	Input	Executor
Thread Scheduler	Thread	Core
Socket Select	TCP Connection	TCP Socket
	UDP Datagram	UDP Socket
CPU Redirect	Network Packet	Core
XDP_SKB	Network Packet	AF_XDP Socket
XDP_DRV	Network Packet	AF_XDP Socket
XDP Offload	Network Packet	NIC RX Queue

Figure 4. Syrup-supported hooks across the stack along with their inputs and executors.

The operating system makes sure that each Syrup program handles as inputs the datagrams (for the case of UDP) or the connection-establishing SYN packets (for the case of TCP) directed to that set of sockets. For the lower layers of the networking stack, `syrupd` does the heavy lifting, filtering inputs to the correct policy programs, as described in §4.3. In both cases, the policy program receives a set of pointers to the packet and its metadata that it can process to make a scheduling decision. For ghOSt-deployed policies, users need to register threads with the policy through a function call and add them to a map. Then, the thread id and the type of thread state change that occurred are passed as inputs to the scheduling program. There is a significant difference between network stack and thread scheduling. In thread scheduling, the policy selects one of the threads/inputs when an executor/core becomes available, while network stack policies select an executor when an input becomes available.

5 Evaluation

We aim to answer the following questions:

1. Can Syrup be used to express and implement a variety of scheduling policies? (§5.2)
2. Can Syrup be used for cross-layer scheduling? (§5.3)
3. Are Syrup policies portable across different hooks? (§5.4)
4. What are Syrup's overheads? (§5.5)

5.1 Experimental Methodology

5.1.1 Experimental Setup In answering the questions above, we use two different sets of servers, set A and set B. **Set A** has two client and one server machines. Each machine includes two Intel Xeon E5-2630 CPUs operating at 2.3GHz, one Intel 82599ES 10GbE NIC, and runs Linux kernel 5.9 unless otherwise noted. **Set B** has two client and one server machines as well. Each machine includes two Intel Xeon Gold 5117 CPUs operating at 2.00GHz, one Netronome Agilio CX 10GbE NIC, and runs Linux kernel 5.9. We use Set B due to the programmability offered by the Netronome NIC. To avoid performance variability, we only use the physical cores in the same socket as the NIC. We configure the NIC to have a number of RX queues equal to the number of hyperthreads used by the application and map the corresponding interrupts to the hyperthread buddies of the hyperthreads that host application threads.

5.1.2 Applications

RocksDB RocksDB [2] is a popular database developed by Facebook that can handle both point (PUT/GET) and range (SCAN) queries. It is an application that presents different scheduling challenges as its queries vary in duration, complexity, or even storage backend (DRAM vs. Flash). We examine a workload scenario where clients issue a mix of GET and SCAN requests. GETs are very short, having a service time of 10-12μs, while SCANs last for much longer, around 700μs. On a server set A machine, we start multiple RocksDB server threads that receive datagrams on the same port using the SO_REUSEPORT option and use Syrup to implement and inject the scheduling policy that selects the socket (and therefore the thread) that handles each incoming datagram (§5.2). We use an open-loop load generator similar to mutilate [33] that transmits requests over UDP. Moreover, in §5.3, we use Syrup to implement and deploy a policy that schedules RocksDB threads to cores.

MICA To showcase Syrup's ability to take advantage of various scheduling hooks, we use MICA [34] as an example application. MICA [34] is a key-value store that achieves high-performance by partitioning data across cores and using client- and NIC-side request steering to minimize data movement. For each key-value request, clients calculate a hash of the key and set the destination UDP port based on that hash. The server inserts flow steering rules into the

NIC that direct incoming requests to queues (and therefore cores). However, this design requires client knowledge of the server's deployment details, an unreasonable requirement for large-scale deployments. If the client is not aware of details about the server's partitions, MICA steers requests to their "home" core in userspace using highly-optimized DPDK queues for communication.

To enable server-side request scheduling in MICA, we built an AF_XDP-based backend. AF_XDP [23] is an address family and socket type which allows the kernel to redirect incoming packets directly to userspace memory buffers and avoid expensive protocol processing. AF_XDP sockets may support two modes of operation depending on the NIC and the kernel version. In the native mode, packets are forwarded to userspace just after the DMA of the buffer descriptor and before the SKB allocation requiring driver support but allowing for zero-copy networking. In the generic mode, packets are forwarded from the `netif_receive_skb()` function after SKB allocation. This mode is driver-independent, but it does not support zero-copy.

When network traffic goes through AF_XDP sockets, we can use Syrup to schedule in two layers, i.e., selecting a NIC RX queue and selecting one of the AF_XDP sockets bound to each queue. For the MICA experiments, we use 8 MICA server threads running on the set B machines and the load generator from the original MICA paper, configured to use our AF_XDP backend.

5.2 Scheduling policies in Syrup

5.2.1 Syrup Makes Scheduling Easy The first goal of our evaluation is to show that Syrup can be easily used for the definition and implementation of a wide variety of scheduling policies that can improve workload performance in practice. Towards that goal, we use a RocksDB deployment with six server threads where 99.5% of the requests that clients issue are GETs and 0.5% are SCANs, similar to the one used by Shinjuku [27]. Head-of-line blocking effects make it challenging to achieve high performance in this workload scenario. Since SCANs constitute less than 1% of the requests issued, the overall 99% latency should solely be determined by the GET latency. However, in Figure 6, we see that this is not the case for the Vanilla Linux policy. The 99% latency for this workload measured on the client side is both very high (>1000μs) and very noisy even for low request rates; the vertical lines represent the standard deviation of the tail latency across five runs. The noise exists because - similarly to Figure 2 - there is load imbalance across network sockets exacerbated by the high skew in the request execution times.

To avoid this imbalance, we use Syrup to implement a simple round-robin policy in less than 10 lines of code (Figure 5a). The same policy is used for the GET-only workload presented in Figure 2. In our policy, we initialize an index (line 1) and increment it every time we schedule a datagram (line 5). The non-atomic increment can lead to benign race conditions, e.g., scheduling two consecutive datagrams to the same socket, that do not affect the policy's performance. Then, we select a socket/thread using the index, ensuring perfect load balancing. In our case, `NUM_THREADS` is a compile-time parameter, but it can alternatively be read dynamically from a Map at run time. Figure 6 shows that the Round Robin Syrup-defined policy eliminates noise and achieves a throughput 124% higher than that of the Vanilla Linux policy before the tail latency explodes.

Nevertheless, we observe that even for the Round Robin policy, SCANs still determine the overall tail latency, i.e., it is more than 1000μs. The latency is so high because shorter GETs can get stuck behind longer SCANs waiting to be served. To avoid this problem, we implemented a SCAN_avoid policy in Syrup which has both a userspace (Figure 5b) and a kernel (Figure 5c) component. For each incoming packet, the scheduling code that runs in the kernel iterates over network sockets until it finds one that is not currently serving a SCAN (lines 10-12). Each iteration selects a random network socket to check (lines 5-6) to avoid imbalance issues. Userspace code (Figure 5b) sets the request type each socket is handling by updating a Map every time it starts (lines 2-3) and finishes (lines 5-6) processing a SCAN request. Syrup allows us to implement this policy that requires userspace-kernel communication in less than 25 lines of code in total. In Figure 6, we see that, by using this SCAN Avoid policy, we can keep the 99% latency at less than 150μs, i.e., 8× lower than Vanilla Linux's latency, for a load as high as 150,000 RPS. At higher loads, the tail latency gradually increases as it is more likely that all sockets are handling SCAN requests, and therefore it becomes harder to avoid head-of-line blocking.

Finally, we use Syrup to develop a third policy (Figure 5d) that peeks into the packet content and makes a more informed scheduling decision. We implement a simple version of the SITA (Size Interval Task Assignment) policy [20] in Syrup. Our policy first performs a bound check using pointer arithmetic on void * pointers allowed by Clang (lines 5-6). It then checks the request type in each packet (line 9). If the request is a SCAN, we steer it to network socket 0 (lines 11-12). If the request is a GET, we round-robin across the rest of the network sockets (lines 14-15). Before Syrup, the implementation of SITA-like policies required significant development effort [16] or even hardware changes [38]. Figure 6 shows that the Syrup-based SITA policy can keep the tail latency low (<150μs) for a load as high as 310,000 RPS, i.e., more than 100% higher than the SCAN Avoid policy. The per-core performance of Syrup for this workload is on par with that of kernel-bypass systems such as Shinjuku [27]. A direct comparison was impossible as our setup lacks the posted interrupt feature needed by Shinjuku.

Conclusion Syrup allows us to quickly implement various scheduling policies, including ones that span across

```c
uint32_t idx = 0;
uint32_t schedule(void *pkt_start,
                  void *pkt_end) {
  idx++;
  return idx % NUM_THREADS;
}
```

(a) Round Robin eBPF

```c
Request * req = parse_request(pkt);
if (req->type == SCAN)
  map_update(&scan_map, &tid, SCAN);
// Do processing...
if (req->type == SCAN)
  map_update(&scan_map, &tid, GET);
```

(b) SCAN Avoid Userspace

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

(c) SCAN Avoid eBPF

```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_THREADS - 1)) + 1;
}
```

(d) SITA eBPF

Figure 5. Scheduling policies implemented in Syrup.

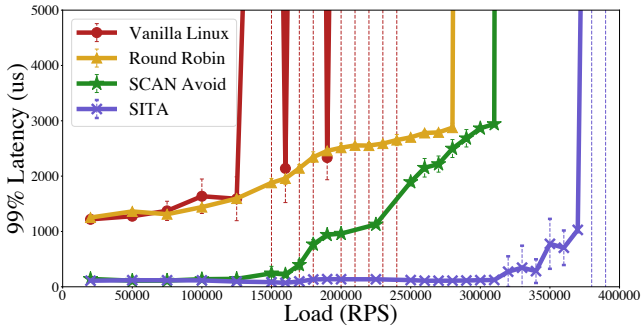
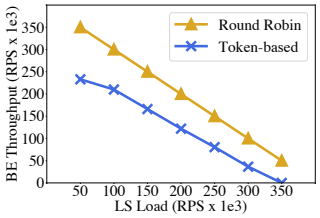
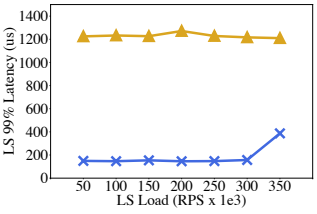


Figure 6. Performance of a RocksDB workload running on 6 cores and serving 99.5% GET - 0.5% SCAN requests when using different Syrup scheduling policies.



(a) BE Throughput (RPS)



(b) LS 99% Latency (us)

Figure 7. Performance of the Round Robin and Token-based schedulers for a workload which scans across different LS and BE load breakdowns. The total offered load (LS + BE) is always 400K RPS.

layers or peek into the inputs, in few lines of code (~10) with little effort, drastically improving application performance (up to 8×).

5.2.2 Using Syrup for more complex policies We also evaluate a more refined version of the token-based scheduling policy first described in §3.4. In the benchmarked scenario, we have two users, one with high-priority latency-sensitive (LS) traffic and the other with lower-priority best-effort (BE) traffic. They both issue RocksDB GET requests. Our token-based policy periodically, i.e., every 100μs, generates tokens the LS user consumes every time one of her requests is served. After each epoch, any leftover tokens are gifted to the BE user. Our token generation rate is 350,000 per second, which guarantees no tail latency explosion in our 6-core setup.

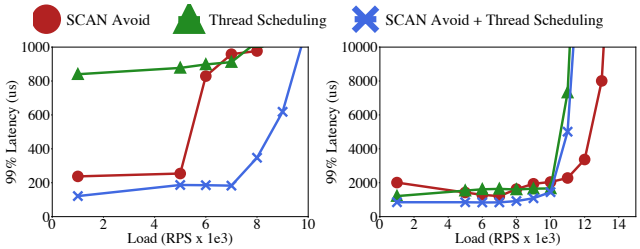
For our experiment, we keep the total offered load constant at 400K RPS, i.e., slightly higher than the saturation point, and scan across different LS and BE load combinations. Figure 7a shows that when the LS load is low, the BE user can take advantage of the spare tokens serving most of her offered load. In Figure 7b, we see that donating tokens to the BE user does not affect the performance of the LS user until her load reaches the saturation point of 350K RPS. If we use the round-robin policy from Figure 5a instead, the BE user achieves a slightly higher throughput but at the cost of 6× higher tail latency for the LS user.

Conclusion Syrup can be used to enforce application-specific Quality-of-Service (QoS) guarantees across the stack.

5.3 Cross-layer scheduling using Syrup

One of the main contributions of Syrup is that it enables coordinated cross-layer scheduling. We showcase that by using Syrup to implement and deploy scheduling policies for a RocksDB workload where 50% of the requests are GETs and 50% are SCANs at two different layers. We configure RocksDB to run with 36 threads on six cores to avoid excessive intra-socket head-of-line blocking and use Linux kernel version 4.19 to be compatible with ghOSt. First, we implement a thread scheduling policy that matches runnable threads to logical cores and use ghOSt to deploy it. Similar to the policy used in Shinjuku [27], our policy gives strict priority to threads processing GET requests, preempting at will threads processing SCAN requests. The policy reads an application-populated `Map` to determine which threads are processing GET requests and which are processing SCANs

and then matches threads to logical cores. We also use the SCAN Avoid policy from §5.2 for request/datagram scheduling at the socket selection layer without statically reserving threads for each request type. It is worth noting that when thread scheduling is active, only five cores can be used for application processing; one is reserved for the spinning ghOSt agent.



(a) GET 99% Latency **(b)** SCAN 99% Latency

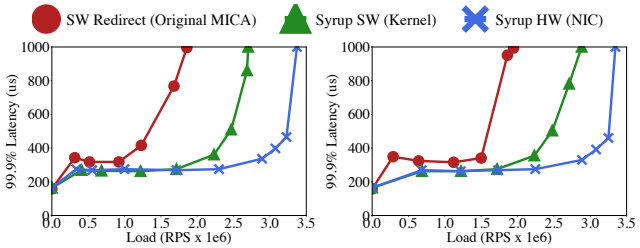
Figure 8. Performance of a RocksDB workload with 50% GET and 50% SCAN requests when Syrup is used for scheduling at one or more layers of the stack. The baseline Linux policy is omitted as its latency is outside the range of our plot.

Figure 8 presents the performance of each of the policies individually as well as when they work together. First, we observe that when only thread scheduling is used, the GET tail latency is very high (>800μs) even for very low load as GETs can still get stuck behind SCANs in a network socket. Similarly, when only the SCAN Avoid policy is used for request scheduling, the GET latency – despite being low initially – explodes at a load of 6000 RPS. This is happening because all cores might be busy running threads serving SCAN requests. The default Linux CFS scheduler, being oblivious to the request handled by each thread, does not preempt them when a thread serving a GET becomes runnable. Syrup allows us to combine the two policies, avoid intra-socket head-of-line blocking and thread scheduling delays, and achieve sub 500μs tail latency for a load as high as 8000 RPS, i.e., 60% higher than the best-case single-layer scheduling scenario. Similarly, combining request and thread scheduling provides lower tail latency for SCANs because it avoids queuing more than one SCANs in the same core or network socket. However, the maximum supported throughput for SCANs (Figure 8b) is slightly lower when thread scheduling is active as one of the cores has to be used by the scheduling agent.

Conclusion Syrup allows us to easily implement policies that span across layers and communicate with each other, maximizing performance.

5.4 Syrup using different hooks

Our main goal when scheduling for MICA is to optimize locality. Using Syrup, we can deploy custom scheduling policies in multiple hooks across MICA's networking stack. The policy deployed in the kernel AF_XDP hook using eBPF, which we call `Syrup SW`, reads the key hash from each incoming packet and selects the corresponding AF_XDP socket of the packet's "home" thread using a simple modulo operation. In this scenario, each MICA thread creates 8 AF_XDP sockets, one for each NIC RX queue. The same policy is reused in the second hook, called `Syrup HW`, this time running on the Netronome NIC and selecting an RX queue. Each MICA thread only creates a single AF_XDP socket bound to a NIC RX queue for this scenario. The scheduling code is very similar to the simple hash-based example discussed in §3.3.



(a) 50% GET - 50% PUT **(b)** 95% GET - 5% PUT

Figure 9. MICA performance for two different workload mixes when scheduling takes place at different layers of the stack. Software redirection was used by MICA originally, when client-side scheduling was impossible.

Figure 9 presents the 99.9% latency for two different MICA workloads, one with 50% GET and 50% PUT operations and one with 95% GET and 5% PUT operations. We observe that the tail latency of the original MICA version that uses application-layer packet redirection exceeds 1ms at a load of 1.7-1.8 MRPS. The MICA version using Syrup for scheduling in the AF_XDP layer (Syrup SW) exceeds that threshold at a load of 2.7-2.8 MRPS. This happens because packet redirection at the application layer may require 2 data movements, one from the core that handles the NIC RX queue to a MICA core that parses the request and one to the request's "home" core. Using Syrup, the MICA core to MICA core communication never needs to happen; each packet is always steered to its "home" MICA core from the core that handled the network interrupt. Finally, the tail latency for Syrup HW explodes at a load of 3.2-3.3 MRPS, i.e., 18% higher than the Syrup SW variation and 83% higher than the version that does not use Syrup. Syrup HW eliminates all end-host data movement since packets are steered directly to the hyperthread-buddy of the MICA core that eventually processes them.

The tail latency across all three variations is relatively higher, and the throughput is relatively lower compared to the numbers reported in the MICA paper [34] because the programmable Netronome NIC we use does not support zero-copy. However, when we tried our AF_XDP backend (Syrup SW) in a non-programmable Intel 82599 NIC that supports the zero-copy XDP_DRV hook, we achieved latency and

throughput similar to those in the MICA paper when the same number of NICs was used.

Conclusion Policies written using Syrup are portable since they can be deployed in different layers of the stack depending on the capabilities of the underlying operating system and hardware.

5.5 Syrup's Overheads

There are two primary sources of overhead associated with Syrup scheduling policies. The first is the CPU cycles consumed to make and enforce a scheduling decision. Table 2 shows the number of lines of code (LoC), x86 instructions, and cycles for each of the policies described §5.2. We can see that all policies are implemented in very few lines of code. The SCAN Avoid policy requires more instructions than the other policies due to loop unrolling. Furthermore, all policies run in less than 2,000 cycles, showing that Syrup can be used for low-latency and high-performance workloads. There is little variation across policies because most of this time is spent on enforcing, e.g., redirecting a packet, rather than making, e.g., choosing a socket, each scheduling decision.

Policy	LoC	Instructions	Cycles (± stdev)
Round Robin	6	56	1563 (± 89)
SCAN Avoid	21	311	1709 (± 115)
SITA	16	81	1699 (± 210)
Token-based	45	106	1582 (± 54)

Table 2. Overhead of different Syrup policies.

Backend	Get (nsec)	Update (nsec)
Host	986	1009
Host Contended	1009	1041
Offload	23735	25001
Offload Contended	25001	24115

Table 3. Map operation latency for different backends.

The second main overhead of Syrup is the time needed to pass information across layers, i.e., to read from and write to a `Map`. Table 3 shows the overhead for different Map operations issued from the userspace for different backends. Accessing a host-based `Map` with 1M elements takes about 1μs regardless of contention (two threads issuing operations concurrently) and type of operation, while accessing a `Map` based on a Netronome NIC (Offload) takes about 25μs. We expect the access cost to the offloaded map to decrease with the adoption of new, faster I/O standards such as CXL [8]. The access cost from Syrup code running in the kernel or on NIC hardware is the same as that of a regular memory access. It is possible for userspace applications to memory-map eBPF maps and access them directly using regular memory operations. However, this feature is only available in newer kernels, so it is not compatible with ghOSt, and we do not use it in our experiments.

6 Discussion

6.1 Adding more backends

Recent research has shown that improving IO request scheduling leads to significant performance gains [22, 30]. One natural extension for Syrup's scheduling model is storage; we can use Syrup to match IO requests with storage device queues. In fact, the token-based policy we evaluate in §5.2 is very similar to the one used by Reflex [30] for IO request scheduling in flash devices.

Syrup's current implementation focuses primarily on end-host scheduling. However, scheduling occurs across the data center stack, from cluster managers and software load balancers to programmable switches. We can extend Syrup to support such backends as they are fully compatible with Syrup's matching view of scheduling; similar to end-host components, they schedule inputs (jobs/requests/packets) to executors (servers). Developers will implement their preferred scheduling in C code that will run in userspace or in P4 code that will run in programmable network devices, and the Syrup framework will deploy that code safely and efficiently across the data center stack. For new hardware devices to play well with Syrup, they need three things: programmability, a matching abstraction between inputs and executors, and support for a Map abstraction which can either reside in the device, the end-host, or remotely.

Extending Syrup to a distributed setting will require solving some interesting research challenges. First, we need to make sure that the Map abstraction still works, allowing Syrup components deployed to different layers of the stack to communicate seamlessly. Potential solutions range from adopting a managed Key-Value Store service to using partitioned systems like Anna [48]. Second, Syrup must guarantee isolation between user-defined P4 programs that run in the same programmable network device. Syrup can enforce isolation by inserting P4 match/action rules that, e.g., use the IP address/port number pair of a network packet to steer it to the correct handling function.

6.2 Making coding in Syrup easier

Even though the safe C subset we use provides a natural way to write and safely inject scheduling functions across the stack, it is sometimes cumbersome to use. For example, developers need to explicitly check before dereferencing a potentially unsafe memory access. We view adding such checks automatically as a natural next step for Syrup.

If we realize the extensions in §6.1, we may end up with a fragmented environment where some of the scheduling code is compiled to eBPF and some to P4. The existence of a P4 to eBPF compiler [5] means that we can avoid such

fragmentation by adopting P4 as the main scheduling language for Syrup. However, given P4's limited match/action model, it will be interesting to explore if it is possible to compile a high-level language similar to Lucid [46] to low-level eBPF and P4 code or even develop a custom domain-specific language for scheduling.

6.3 Support for Late Binding

Most of the hooks Syrup uses across Linux's networking stack only support early binding of inputs to executors. In other words, the arrival of a packet triggers a scheduling function that has to select an executor at that time. While early-binding is convenient as it does not require scheduler-side queueing, it can lead to head-of-line blocking when inputs with short processing time get stuck behind ones with longer processing time in the same executor [31, 42, 51]. This problem can be avoided through late binding. When late binding is used, an input is assigned to an executor only when the executor can immediately start to process it [27, 41]. Implementing late binding in the Linux networking stack requires storing packets in a temporary buffer and triggering the scheduling function when an executor signals it is available, e.g., when a thread calls `recvmsg` on a socket.

6.4 Scheduling Streams

Scheduling requests sent over streams is particularly challenging. This is why recently-developed network protocols [39] and RPC frameworks [31] are request-based. However, it is possible to support scheduling requests sent over a TCP stream using a Linux mechanism, the Kernel Connection Multiplexor (KCM) [10]. With KCM, users can programmatically identify request boundaries across packets in TCP streams and do request-level scheduling. For the case of multi-packet requests over UDP, users can store a mapping of requests to executors in a Map, similar to what [51] did.

7 Related Work

Cross-layer scheduling There is a large body of work on scheduling across different layers of the stack. Some systems prefer to do scheduling as early as possible. For example, R2P2 [31] offloads request scheduling to a programmable switch. Other systems exploit advanced features of Network Interface Cards (NICs) for scheduling. IX [14] and eRPC [28] use Receive Side Scaling to spread packets to cores based on the 5-tuple hash. Mind the Gap [24] implements a centralized scheduling policy on ARM cores present in modern NICs. Reflex [30] uses a token-based policy for multi-tenant SSD deployments. ZygOS [42] implements work-stealing on top of IX to reduce load imbalance across cores. Shinjuku [27] uses a dedicated core for scheduling and low-overhead interrupts for fast preemption. TAS [29] dynamically adapts the number of cores used for network processing to be proportional with the load and selects userspace contexts that handle incoming connections or requests. RackSched [51] implements least-loaded request-to-server scheduling in a programmable switch while using a policy similar to Shinjuku's in each end-host. Finally, nanoPU [26] implements both core selection and hardware thread scheduling in hardware. Syrup makes it easy to develop and deploy policies such as those introduced by these systems without implementing a new system for each.

User-informed kernel decisions Passing application-specific information from userspace to the kernel and vice versa has been around at least since the early 90s and scheduler activations [11]. Redline [50] allows interactive applications to declare specifications that the kernel's resource managers use to ensure responsiveness. Recently, MittOS [19] allowed applications to pass deadline information to the kernel so that requests that will fail to meet the deadline get rejected quickly.

Syrup is more directly related to systems that give applications safe control of functionalities traditionally handled by the kernel. The first such design was the Exokernel [17] that used a small kernel to export hardware resources to untrusted library operating systems through a low-level interface. Exokernel's capability of safely "downloading" application code into the kernel, e.g., packet filters, was an early inspiration for the development of the eBPF framework. Infokernel [12] exposed kernel information to the userspace, which can, in turn, change parameters and inputs to manipulate policies used by the kernel. Finally, recently-developed Caladan's [18] userspace code communicates with a custom kernel driver to drive interference-aware scheduling decisions. Syrup exploits eBPF to safely inject code to a monolithic Linux kernel while allowing the exchange of arbitrary information between the user- and kernel-space through the general Map abstraction.

eBPF Even though it is a relatively new kernel feature, eBPF has seen significant industry adoption. Cloudflare uses eBPF extensively for DoS mitigations and layer 4 load balancing [4]. Cilium [7] has built its entire business model around eBPF, offering an eBPF-backed networking, observability, and security platform on top of Kubernetes. The industry use case more closely related to Syrup is Facebook's Katran [9] which is a C++ library and BPF program for building high-performance L4 load balancing forwarding planes. Unlike Syrup, Katran provides support for only a single scheduling decision (L4 packets/connections to hosts) in a single layer of the stack (XDP) while it does not allow different applications to safely specify and run their own policies on the same server.

Academic projects have very recently started to use eBPF in more exotic ways. BMC [6] exploits eBPF to create an in-kernel memcached cache achieving better efficiency and performance even than kernel-bypass techniques. Prism [21] uses eBPF and a connection hand-off protocol to implement efficient proxies for object storage systems. Finally, Wu et al.

explore the potential use of eBPF for storage [49], identifying challenges and opportunities.

8 Conclusion

We propose Syrup, a framework for user-defined scheduling. Syrup allows application developers to specify scheduling policies in a safe, efficient, and high-level manner across the stack. Syrup's design treats scheduling as a matching problem between work units and executors without burdening developers with low-level details and system mechanisms. We show that applications can use Syrup to unlock performance gains that previously required months of development work in a few lines of code.

Acknowledgments

We thank Peter Kraft, Qian Li, John Ousterhout, Deepti Raghavan, Amin Vahdat, Thomas Wenisch, Eric Brewer, our shepherd Ryan Stutsman, and the anonymous SOSP reviewers for their helpful feedback. This work is partially supported by Stanford Platform Lab sponsors and Facebook.

References

[1] 2010. rfs: Receive Flow Steering. https://lwn.net/Articles/381955/.

[2] 2012. RocksDB. https://rocksdb.org/.

[3] 2017. A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[4] 2019. Cloudflare architecture and how BPF eats the world. https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/.

[5] 2020. p4c-ubpf: a New Back-end for the P4 Compiler. https://p4.org/p4/p4c-ubpf.

[6] 2021. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/ghigoff

[7] 2021. Cilium. https://cilium.io/.

[8] 2021. Compute Express Link: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/about-cxl.

[9] 2021. Katran. https://github.com/facebookincubator/katran.

[10] 2021. Kernel Connection Multiplexor. https://www.kernel.org/doc/Documentation/networking/kcm.txt.

[11] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1991. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *SIGOPS Oper. Syst. Rev.* 25, 5 (Sept. 1991), 95–109. https://doi.org/10.1145/121133.121151

[12] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James A. Nugent, and Florentina I. Popovici. 2003. Transforming Policies into Mechanisms with Infokernel. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 90–105. https://doi.org/10.1145/1165389.945455

[13] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. 2019. RSS++: Load and State-Aware Receive Side Scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT '19)*. Association for Computing Machinery, Orlando, Florida, 318–333. https://doi.org/10.1145/3359989.3365412

[14] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[15] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPC-Valet: NI-Driven Tail-Aware Balancing of μs-Scale RPCs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, Providence, RI, USA, 35–48. https://doi.org/10.1145/3297858.3304070

[16] Diego Didona and Willy Zwaenepoel. 2019. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 79–94. https://www.usenix.org/conference/nsdi19/presentation/didona

[17] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. *SIGOPS Oper. Syst. Rev.* 29, 5 (Dec. 1995), 251–266. https://doi.org/10.1145/224057.224076

[18] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[19] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. 2017. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, Shanghai, China, 168–183. https://doi.org/10.1145/3132747.3132774

[20] Mor Harchol-Balter, Mark E. Crovella, and Cristina D. Murta. 1999. On Choosing a Task Assignment Policy for a Distributed Server System. *J. Parallel and Distrib. Comput.* 59, 2 (1999), 204 – 228. https://doi.org/10.1006/jpdc.1999.1577

[21] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. 2021. Prism: Proxies without the Pain. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/hayakawa

[22] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. 2019. Multi-Queue Fair Queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 301–314. https://www.usenix.org/conference/atc19/presentation/hedayati-queue

[23] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. Association for Computing Machinery, Heraklion, Greece, 54–66. https://doi.org/10.1145/3281411.3281443

[24] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2019. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks (HotNets '19)*. Association for Computing Machinery, Princeton, NJ, USA, 60–68. https://doi.org/10.1145/3365609.3365856

[25] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, Virtual Event, Germany. https://doi.org/10.1145/3477132.3483542

[26] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Nick McKeown, and Changhoon Kim. 2020. The nanoPU: Redesigning the CPU-Network Interface to Minimize RPC Tail Latency. arXiv:2010.12114 [cs.AR]

[27] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[28] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[29] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, Dresden, Germany, Article 24, 16 pages. https://doi.org/10.1145/3302424.3303985

[30] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. Association for Computing Machinery, Xi'an, China, 345–359. https://doi.org/10.1145/3037697.3037732

[31] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 863–880. https://www.usenix.org/conference/atc19/presentation/kogias-r2p2

[32] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. 2020. Provable Multicore Schedulers with Ipanema: Application to Work Conservation. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, Heraklion, Greece, Article 3, 16 pages. https://doi.org/10.1145/3342195.3387544

[33] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. Association for Computing Machinery, Amsterdam, The Netherlands, Article 4, 14 pages. https://doi.org/10.1145/2592798.2592821

[34] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[35] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Association for Computing Machinery, London, United Kingdom, Article 1, 16 pages. https://doi.org/10.1145/2901318.2901326

[36] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Ravichandra Addanki, Mehrdad Khani Shirkoohi, Songtao He, et al. 2019. Park: An Open Platform for Learning-Augmented Computer Systems. *Advances in Neural Information Processing Systems 32 (NIPS 2019)* (2019).

[37] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. Association for Computing Machinery, Huntsville, Ontario, Canada, 399–413. https://doi.org/10.1145/3341301.3359657

[38] A. Mirhosseini, B. L. West, G. W. Blake, and T. F. Wenisch. 2020. Q-Zilla: A Scheduling Framework and Core Microarchitecture for Tail-Tolerant Microservices. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 207–219. https://doi.org/10.1109/HPCA47549.2020.00026

[39] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, Budapest, Hungary, 221–235. https://doi.org/10.1145/3230543.3230564

[40] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[41] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. Association for Computing Machinery, Farmington, Pennsylvania, 69–84. https://doi.org/10.1145/2517349.2522716

[42] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, Shanghai, China, 325–341. https://doi.org/10.1145/3132747.3132780

[43] Alexander Rucker, Muhammad Shahbaz, Tushar Swamy, and Kunle Olukotun. 2019. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019 (APNet '19)*. Association for Computing Machinery, Beijing, China, 71–77. https://doi.org/10.1145/3343180.3343184

[44] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. 2021. *We Need Kernel Interposition over the Network Dataplane*. Association for Computing Machinery, Virtual, USA, 152–158. https://doi.org/10.1145/3458336.3465281

[45] L. Sha, R. Rajkumar, and J. P. Lehoczky. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Comput.* 39, 9 (Sept. 1990), 1175–1185. https://doi.org/10.1109/12.57058

[46] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A Language for Control in the Data Plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, Virtual Event, USA, 731–747. https://doi.org/10.1145/3452296.3472903

[47] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery, Online Event, United Kingdom, 1–16. https://doi.org/10.1145/3447786.3456225

[48] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. 2019. Anna: A KVS for any scale. *IEEE Transactions on Knowledge and Data Engineering* (2019).

[49] Yu Jian Wu, Hongyi Wang, Yuhong Zhong, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. 2021. BPF for Storage: An Exokernel-Inspired Approach. In *Proceedings of the 18th ACM Workshop on Hot Topics in Operating Systems (HotOS '21)*. Association for Computing Machinery.

[50] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2008. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/osdi-08/redline-first-class-support-interactivity-commodity-operating-systems

[51] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. 2020. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1225–1240. https://www.usenix.org/conference/osdi20/presentation/zhu