Draconis: Network-Accelerated Scheduling for Microsecond-Scale Workloads

Sreeharsha Udayashankar
s2udayas@uwaterloo.ca
University of Waterloo
Canada

Ashraf Abdel-Hadi
a32abdel@uwaterloo.ca
University of Waterloo
Canada

Ali Mashtizadeh
mashti@uwaterloo.ca
University of Waterloo
Canada

Samer Al-Kiswany
alkiswany@uwaterloo.ca
University of Waterloo
Canada

Abstract

We present Draconis, a novel scheduler for workloads in the range of tens to hundreds of microseconds. Draconis challenges the popular belief that programmable switches cannot house the complex data structures, such as queues, needed to support an in-network scheduler. Using programmable switches, Draconis achieves the low scheduling tail latency and high throughput needed to support these microsecond-scale workloads on large clusters. Furthermore, Draconis supports a wide range of complex scheduling policies, including locality-aware scheduling, priority-based scheduling, and resource-based scheduling.

Draconis reduces the 99th percentile scheduling latencies by 3×−200× when compared to state-of-the-art software-based and network-accelerated schedulers, on a range of synthetic workloads. Our evaluation also demonstrates that Draconis has 52× higher throughput than server-based scheduling systems.

ACM Reference Format:
Sreeharsha Udayashankar, Ashraf Abdel-Hadi, Ali Mashtizadeh, and Samer Al-Kiswany. 2024. Draconis: Network-Accelerated Scheduling for Microsecond-Scale Workloads. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3627703.3650060

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3650060

1 Introduction

Online services such as real-time analytics [1], financial services [2, 3], algorithmic trading [4, 5], and interactive applications have strict user-facing service level objectives (SLOs) [6, 7]. The underlying systems supporting these services must achieve high throughput and a low tail latency within tens of microseconds [8]. Examples of such systems include microsecond-scale key-value stores [9], multi-core schedulers [10, 11], cluster schedulers [12, 13], web-services [14], and databases [15].

At the heart of these services lie task schedulers. Designing schedulers for large scale clusters that serve tasks in the tens-of-microsecond range is challenging. First, the scheduler needs to achieve *high scheduling throughput* to effectively use the cluster. Second, given the short task duration, the scheduler should have *low tail scheduling latencies.* State-of-the-art schedulers struggle to support these large-scale low-latency workloads. For instance, Chen et al. [16] characterized the overheads in data-analytics frameworks with microsecond-scale workloads and found that scheduling overheads account for nearly 60% of the total execution time. State-of-the-art schedulers fail to meet the stringent requirements for these workloads due to numerous reasons such as node-level blocking [12, 17], low scheduling throughput [13, 18], and sub-optimal scheduling decisions [19–21].

Two recent systems, R2P2 [17] and RackSched, [12] explore using programmable switches [22] to accelerate scheduling decisions. Individual tasks are directly queued to worker nodes. The switch is used to monitor the lengths of each worker queue and forward an incoming request to the worker with the shortest queue i.e. Join-Shortest-Queue (JSQ) scheduling. Both projects use variants of JSQ scheduling; R2P2 uses a Join-Bounded-Shortest-Queue (JBSQ) approach while RackSched uses the power-of-two choices with a JSQ policy.

Both systems suffer from inefficient scheduling as a result of three fundamental shortcomings. First, using the JSQ/JBSQ scheduling policy results in inferior scheduling decisions compared to global single-queue scheduling approaches [12, 23, 24]. Second, using a push-based model to push and queue tasks at worker nodes causes *node-level* *blocking*, i.e., a task may wait at an executor's queue while other executors are available in the cluster. Finally, due to perceived programmable switch limitations, state-of-the-art systems use inefficient techniques such as excessive packet recirculation or sampling.

Previous studies have shown that centralized first come first served (cFCFS) is the optimal scheduling policy for light-tailed workloads [25] and that a single queue approach is better than scheduling from multiple queues [24]. Thus, the aforementioned inefficiencies can be resolved by eliminating worker side queues and hosting a global cFCFS task queue on a switch.

We present Draconis, a novel scheduler design for high throughput and low-latency workloads. To support large clusters and reduce tail latencies, Draconis accelerates the scheduler using modern programmable switches [22, 26]. Draconis overcomes the shortcomings of the state-of-the-art systems by making two fundamentally different design choices. Draconis does not have distributed queues at the worker nodes. Instead, we build a central FCFS queue and host it at a programmable switch. Second, Draconis adopts a pull-based scheduling model. Executors pull tasks from the central queue. This approach eliminates node-level blocking and precisely selects the next free executor for a task, leading to superior scheduling decisions compared to other systems.

We use two novel techniques (§4) to realize this approach. First, we present a novel P4-compatible circular queue design built using *delayed pointer correction*. The design judiciously uses packet recirculation to overcome the memory-access limitations of programmable switches. Second, we present *task swapping* and *queue replication* techniques that allow the implementation of complex scheduling policies on the switch. In addition to FCFS scheduling, we use these techniques to implement two constraint-based policies: data-locality-aware and resource-aware scheduling, and a class-of-service based policy: priority-aware scheduling.

Draconis targets workloads with execution times in the tens to hundreds of microseconds [4, 8, 27, 28], for whom node-level blocking is the primary bottleneck (§2.2). To avoid node-level blocking, Draconis executors pull tasks only when they are free. The worker thread is idle while pulling a task. This approach trades off a small amount of CPU efficiency (§3) to eliminate node-level blocking worth tens to hundreds of microseconds.

Limitations imposed by programmable switch hardware are one of the common concerns associated with building network-accelerated schedulers. Our analysis (§7) shows that Draconis can support task queues of up to one million tasks on modern programmable switches [26]. In addition, Draconis adopts existing mechanisms to handle situation in which tasks (§4.3) or their parameters (§4.4) do not fit in a single packet.

Our evaluation of Draconis on a cluster with a Barefoot Tofino switch [22] shows significant performance benefits for both synthetic and real-world workloads (§8). Our evaluation shows that the 99th percentile of Draconis' scheduling delay is lower by 3×, 120×, 200× and 20× when compared to RackSched [12], R2P2 [17], Sparrow [19] and Draconis-DPDK (a DPDK-based [29] Draconis implementation), when running a workload with 500μs tasks. Draconis also achieves 52×–116× higher scheduling throughput over server-based schedulers such as Draconis-DPDK and Sparrow [19]. Draconis's source code is available on GitHub [30].

2 Background and Motivation

2.1 Programmable Switches

Programmable switches facilitate the implementation of an application specific packet-processing workflow that is executed at line speed. They contain multiple hardware pipelines, each of which is composed of multiple stages. Packets go through the pipeline serially, stage by stage.

Stages use match-action tables. If a packet matches a rule in a table, the corresponding action is executed. Each stage has its own dedicated resources, including tables and register arrays (memory buffers). Stages can share data through the packet headers and small per-packet metadata propagated between stages as the packet proceeds through the pipeline. Each stage processes a single packet and different stages within the pipeline process different packets at the same time.

2.1.1 Challenges. The need to execute custom actions at line speed restricts what modern programmable switches can do. Modern switches limit (1) the number of stages per pipeline, (2) the number of tables and registers per stage, (3) the number of times any register can be accessed per packet, (4) the amount of data that can be read or written per packet per register, and (5) the size of the per-packet metadata passed between stages. In addition, modern switches lack support for loops or recursion.

The restrictive memory model constitutes a particular challenge to building an in-network scheduler. As providing access to registers from multiple stages (and thus packets) can result in read-write hazards, a register can only be operated on once per packet within all modern switches [31]. The operation can either be a simple read/write or an arithmetic operation (e.g., read-and-increment).

Thus, implementing a task queue is complicated, because queue operations access the queue size variable twice: once to check whether the queue is empty or full and a second time to increment or decrement it. This has led to the popular belief that building dynamic data structures such as queues is not possible on programmable switches [12].

2.2 Network-Accelerated Scheduling

R2P2 [17] and RackSched [12] explored using programmable switches to accelerate scheduling for a cluster of workers. Each worker hosts multiple executors. These schedulers do not maintain a queue of tasks on the switch. Instead, each worker node (RackSched) or executor (R2P2) has its own queue of tasks. Both schedulers use variants of the Join-Shortest-Queue (JSQ) scheduling policy to schedule a task on an executor with the shortest queue.

R2P2 [17] uses a Join-Bounded-Shortest-Queue (JBSQ) policy with a bounded queue size k (typically 3) on each executor. To implement JBSQ, R2P2 maintains an array of counters at the switch to track the queue size of each executor. When a new task is received, R2P2 tries to find an executor whose queue size is zero. However, the challenging switch programming model requires R2P2 to use up to $O(n)$ packet recirculations where n is the number of executors. If no executor with a queue size of zero is found, the switch uses another $O(n)$ recirculations to find an executor with a queue size of one and so on. In total, R2P2 uses $O(n \times k)$ recirculations in the worst case. If all executor queues are full, R2P2 keeps recirculating a task until a spot becomes free at one of the executor queues. If a switch does not have the capacity to recirculate a packet, it simply drops it. As shown in §8.3, this can lead to dropping many tasks in bursty workloads, which drastically increases scheduling tail latency.

RackSched [12] approximates a Join-Shortest-Queue (JSQ) policy by maintaining queues at each worker node (not per-executor). Each worker node hosts multiple executors. RackSched uses JSQ at the switch and an additional intra-node scheduler to schedule tasks between executors within a worker node. RackSched advises using an intra-node cFCFS policy without preemption for light-tailed workloads. For heavy-tailed workloads, they use an intra-node *Processor Sharing* policy with preemption [12] to avoid *head-of-line blocking*, i.e., shorter tasks being blocked behind long running tasks

The queue lengths are tracked on the switch. To avoid excessive recirculation while looking for the shortest queue, RackSched uses the power-of-two choices and samples the queue lengths of two nodes. The scheduler forwards the task to the shorter queue. Sampling achieves sub-optimal scheduling decisions at high loads because it is not guaranteed to select the shortest queue across the cluster. In addition, the intra-node scheduler adds extra scheduling overhead, worsening tail latency (§8).

2.2.1 Node-Level Blocking Problem.

Scheduling systems adopting distributed queue designs suffer from *node-level blocking*, i.e. a task may be stuck waiting in a worker node's queue even when executors on other worker nodes are free. Preemption does not address the node-level blocking problem. RackSched and R2P2 do not implement work stealing across nodes to address node-level blocking, as this would incur additional coordination overhead and network transfer latencies.

2.2.2 Draconis' Approach.

The design choices made by R2P2 and RackSched are a consequence of being unable to build a centralized task queue on programmable switches. Previous studies show that centralized global task queues are more efficient than the best JSQ policies [23]. Draconis follows a fundamentally different design approach by proposing a novel P4-compatible circular queue design to hold tasks at the switch, eliminating the need for executor-side queues.

2.3 Server-based Scheduling

Existing server-based cluster schedulers adopt one of two design paradigms: centralized or distributed.

2.3.1 Centralized Schedulers.

Within the centralized scheduling paradigm, a single scheduler is used to make scheduling decisions for the entire cluster. Some examples of centralized schedulers include Firmament [13] and the Spark Native Scheduler [18]. Despite achieving optimal task placement, server-based centralized schedulers are unable to achieve high scheduling throughputs while maintaining low tail latencies, because they are bottlenecked by the performance of a single node.

For instance, Gog et al. [13] report that Firmament cannot scale to more than 100 nodes with 6 physical cores each (1200 executors total), when running 5 ms tasks. Similarly, in our experiments, the Spark Native Scheduler [18] could not handle sub-second tasks, which confirms a similar observation made by the authors of Sparrow [19]; The scheduling delay at 50% cluster utilization was 3 seconds, when running tasks with a mean service time of 500 µs. Above 50% utilization, the scheduler could not keep up and experienced infinite queueing.

2.3.2 Distributed Schedulers.

Distributed scheduling utilizes multiple schedulers to make scheduling decisions for the cluster, to address the scalability problem associated with centralized server-based schedulers. However, this often results in sub-optimal scheduling decisions. For instance, Sparrow [19] and Hopper [20] use the power-of-k-choices approach to select executors for scheduling, which is inaccurate at high cluster loads. Apollo [21] uses a centralized metadata service that only periodically monitors the cluster nodes and hence has stale data.

3 Draconis Overview

Draconis is an in-network centralized scheduler that precisely assigns tasks to free executors with minimal overhead. An executor is a process running on a worker node which pulls tasks from the scheduler to execute. Typically, worker nodes run multiple executors, one on each available logical core (i.e., hardware thread).

Figure 1 shows Draconis architecture with clients, worker nodes, and a programmable switch. Clients submit a task or a batch of independent tasks (1) which are queued on the programmable switch (2). When an executor is free, it sends

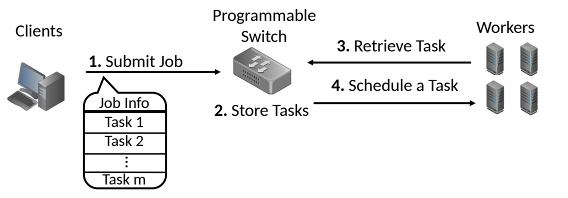


Figure 1. Draconis' Architecture

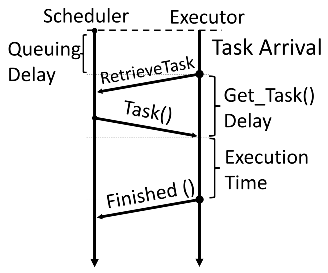


Figure 2. Draconis' scheduling timeline

a task request (3) to the switch, which then schedules a task for it to execute (4).

State-of-the-art network-accelerated schedulers [12, 17] follow a push-based approach and queue tasks at workers leading to node-level blocking. Draconis avoids this by using a single switch-based global queue and pull-based scheduling. Draconis targets workloads with task execution times in the tens to hundreds of microseconds [4, 8, 27, 28]. For these workloads, Draconis presents a good trade off by eliminating node-level blocking worth tens to hundreds of microseconds, at the cost of a single RTT worth of CPU efficiency. Modern network advances promise sub-microsecond RTTs [32] which will further reduce this overhead.

3.1 Draconis Executors and Clients

Draconis Client. Draconis adopts the client model supported by schedulers such as R2P2 [17], Sparrow [19] and the Spark Native Scheduler [18]. The Draconis client within our implementation can submit single or batches of independent tasks. Draconis supports request batching to readily integrate with data processing frameworks such as Spark. These frameworks contain call-graph optimization modules, which track data dependencies and submit batches of independent tasks to a scheduler.

Executors. When an executor becomes free, it sends a message to the switch to request a new task. The switch retrieves a task from the switch task queue and sends it back to the executor. This approach effectively eliminates node-level blocking, as tasks are sent precisely to the next free executor available to run them. The executor finishes the task and sends a completion response back to the client via the scheduler. The request for a new task is piggybacked on this completion response. If the scheduler has no tasks available, it sends a no-op task to the executor, which sends another task request periodically.

The executor is idle for a single RTT (typically a few microseconds) while retrieving a task. This represents a small loss of efficiency in executor usage (less than 3% when running 100 µs tasks). The benefits of eliminating node-level blocking far outweigh this loss in efficiency.

3.2 Programmable Switch

Draconis hosts the centralized in-network scheduler on a programmable switch [22, 26]. The scheduler holds tasks in the switch memory until an executor is available. To do this, the scheduler adds these tasks to a circular queue along with the identifiers (IP address and port) of the submitting client. When an executor asks for a task, the scheduler assigns a task to it based on the scheduling policy.

Despite its simplicity, implementing this design on modern programmable switches is challenging due to their restrictive programming and memory model.

Deployment. Draconis can be deployed within the top-of-the-rack switch in rack-scale computing platforms. If Draconis is deployed on multi-rack clusters, then similar to the state-of-the-art projects that use programmable switches [33–36], the network controller installs forwarding rules to forward all job-submission requests through a single switch, which runs the Draconis scheduler. The controller typically selects a common ancestor switch of the cluster nodes. While this approach may create a longer path than traditional forwarding does, the effect of this change is minimal. Li et al. [34] reports that for 88% of cases, this approach does not increase the request latency.

3.3 Fault Tolerance

On switch failure, a new switch is selected to run the scheduling pipeline. Clients will time out on all previously submitted tasks and resubmit them.

Similar to the state-of-the-art schedulers such as R2P2 [17] and Sparrow [19], if a task fails (due to executor or communication failures), this failure is exposed to the client. This allows the client to handle the failure in an application-specific fashion, for instance by resubmitting these tasks to the scheduler.

4 System Design

We first present the building blocks required to implement an in-network scheduler including the network protocol and the task queue. Following this section, we discuss how these components are used to design schedulers for a wide range of policies, including FCFS, priority-based, resource-constrained, and data locality-aware scheduling.

4.1 Network Protocol

Draconis introduces an application-layer protocol embedded in a packet's L4 payload. Similar to existing schedulers [12, 17], our system uses UDP to reduce latency and simplify the scheduler design. Draconis introduces two new packet types: *job_submission* packets submit a batch of tasks to the scheduler and *task_assignment* packets send a task to an executor. A single job consists of one or more *job_submission* packets. Figure 3 shows the main fields of the *job_submission* packet:

- `OP_CODE`: The request type indicating this is a job submission.
- `UID, JID`: The user and job IDs.
- `#TASKS`: The number of tasks in a packet. The scheduler uses this field to parse the job-submission packet.
- A list of `TASK_INFO` metadata for the tasks in the job.

The task information (`TASK_INFO`) includes the following:

- `TID`: The task identifier within a job. The tuple `<UID, JID, TID>` is a unique identifier for any task in the system.
- `FN_ID, FN_PAR`: The identifier and arguments of the pre-compiled function. This task information is similar to that required by state-of-the-art systems [17–19].
- `TPROPS`: The policy-specific properties used for scheduling this task. This field can hold information related to task priority or data locality.

To assign a task to an executor, the scheduler sends it a *task_assignment* packet. The packet headers contain the task information as well as the client's information (IP address and port number).

When other types of packets are encountered, Draconis adopts the functionality of a regular switch and simply forwards them to their destination. This makes Draconis safe for colocation with other protocols.

4.2 Circular Queue Design

Draconis stores tasks in a circular queue using switch registers. Each queue entry contains the task information (`TASK_INFO`) along with the associated client information. The circular queue has two 32-bit pointers: `add_ptr` and `retrieve_ptr`. The `add_ptr` points to the next empty queue entry where a new task can be inserted, while the `retrieve_ptr` points to the next task to be scheduled.

In traditional circular queue implementations, to enqueue a new task, one typically checks whether the queue is full by computing the difference between the pointers. If the queue is not full, then the new task is added to the queue and `add_ptr` is incremented. However, this design cannot be implemented on current switches because it accesses `add_ptr` twice; it first checks the pointer, then possibly increments it. The dequeue operation faces a similar challenge. Another plausible design is to check the queue size before adding or retrieving a task. This approach accesses the queue size

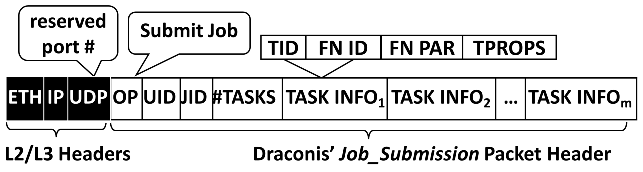


Figure 3. Draconis' *job_submission* packet.

twice as well and cannot be directly implemented on modern switches.

To work around this limitation, Draconis uses an atomic `read_and_increment(add_ptr)` operation to read and increment `add_ptr` in one access, because it can access this pointer only once per packet. It then checks whether the queue is full by computing the difference between the `retrieve_ptr` and `add_ptr`. If the queue is not full, then Draconis uses the `add_ptr` value to add a task to the queue. However, this approach increments `add_ptr` even when the queue is full. Similarly, to dequeue a task, Draconis calls `read_and_increment(retrieve_ptr)` and increments `retrieve_ptr` even when the queue is empty. In these cases, the pointers must be corrected, albeit in a future packet because the pointers can only be accessed once per packet. We discuss how to detect and correct incorrect pointers later in section (§4.5).

4.3 Job Submissions

The client submits a job by populating the headers of a *job_submission* packet (Figure 3) and sending the packet to the scheduler. The scheduler then enqueues the job's tasks. The packet may contain one task or a set of tasks.

Two switch limitations complicate adding a set of tasks to the queue: modern switches do not permit loops or recursion, and the scheduler can access a register (the queue) only once per packet. To work around these limitations, Draconis checks the number of tasks field (`#TASKS`) within a packet. If it is larger than zero, then it removes the first task from the list of tasks in the packet, calls `read_and_increment(add_ptr)`, and adds the task.

Adding Multiple Tasks. If the *job_submission* packet (Figure 3) contains multiple tasks, Draconis uses packet recirculation (i.e., the ability to resubmit a packet from the egress pipeline to the ingress pipeline and reprocess it as a new packet). The scheduler removes the first task from the packet, adds it to the queue, decrements the `#TASKS` field, and recirculates the packet. Draconis continues to recirculate the packet until all tasks have been added to the queue.

Handling a Full Queue. When enqueuing a new task, the scheduler calls `read_and_increment(add_ptr)`, then compares `add_ptr` and `retrieve_ptr` to determine whether the queue is full. If the queue is not full, then the scheduler adds the task to the queue. If the queue is full, the scheduler sends an *error_packet* to the client, containing the list of tasks that could not be added to the queue. The client will retry the submission of these tasks after a short wait.

Handling Large Jobs. A few jobs within the application may require the submission of thousands of parallel tasks, which are larger than the size of one *job_submission* packet. Such jobs can be split into multiple *job_submission* packets instead. As the tasks within a job are independent and executed in parallel, this would not affect their execution correctness. Thus, Draconis is not limited by MTU Size.

4.4 Handling Tasks with Large Parameters

Some tasks may have large parameters that do not fit in the fixed-size parameter field (`FN_PAR`) in the packet. Similar to state-of-the-art schedulers, Draconis supports two mechanisms for handling tasks with large parameter sizes. The first approach is similar to R2P2 [17], where the submitted task does not contain the task information or parameters. Instead, the client submits a special transmission function. When this transmission task is scheduled on an executor, the executor would contact the client directly to retrieve the actual task and parameters. The second supported approach is the typical approach used by data analytics engines: clients first store the input data on an in-memory storage system deployed on the same cluster with the executors, then they add a pointer to the input data in the task parameter field. Our locality-based scheduling policy further schedules tasks on nodes that hold the data (§5.3).

4.5 Pointer Correction

When the scheduler receives a *job_submission* packet, it executes `read_and_increment(add_ptr)` first, then checks whether the queue is full. If the queue is full, incrementing `add_ptr` was a mistake. To correct this mistake, the scheduler recirculates a repair packet to reset the pointer to its original value. To avoid a case in which multiple *job_submission* packets try to reset the pointer, we have repair flags on the switch to ensure that the scheduler only recirculates one repair packet.

Similarly, task retrieval operations call `read_and_increment(retrieve_ptr)`, then check whether the retrieved task is valid. If the retrieved task is invalid, indicating that the queue is empty, then incrementing the pointer was a mistake. We delay fixing this pointer until the next *job_submission* packet is received.

When the next *job_submission* request is received, the scheduler adds the first task in the job to the queue. The scheduler then checks whether `retrieve_ptr` needs adjusting (i.e., whether `retrieve_ptr` is larger than `add_ptr`). If `retrieve_ptr` needs adjusting, the scheduler recirculates a packet and sets the pointer to equal the index of the newly added task. To avoid concurrent repairs, repair flags are once again used. We discuss how Draconis handles race conditions related to pointer correction in §4.7.

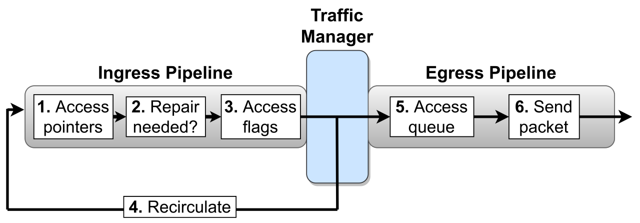


Figure 4. Simplified Draconis pipeline showing the order of operations that deal with queue pointers on the switch.

4.6 Task Retrievals

To avoid node-level head-of-line blocking, executors retrieve tasks only when they become free. To retrieve a task, an executor sends a *task_request* packet to the scheduler. The scheduler calls `read_and_increment(retrieve_ptr)` and reads one task from the queue. If a task satisfying the scheduling policy is available in the queue, the task is sent to the executor. Otherwise, if no valid task is found, then a no-op task is sent to the executor. The executor then repeats the request.

4.7 Handling Race Conditions

A pipeline in a programmable switch processes packets serially. Figure 4 shows the logical order of Draconis' operations that deal with queue pointers. Packets first read and increment the `add_ptr` or `retrieve_ptr` ((1) in Figure 4). *Job_submission* packets then check if there is a need to fix any of the pointers (2). If Draconis detects that the queue is full and that the `add_ptr` has been incorrectly incremented, it sets a flag in (3) and recirculates a repair packet to fix the `add_ptr` (4). Similarly, if Draconis detects that the `retrieve_ptr` is ahead of the `add_ptr`, it sets another flag in (3) and recirculates a repair packet to fix the `retrieve_ptr` (4).

Queue access occurs in the egress pipeline. In (5), *job_submission* packets add tasks while *task_retrieval* packets retrieve tasks from the queue. In (6), *job_submission* packets send an acknowledgment to the client while *task_retrieval* packets send a task or a no-op to the executor.

With the pipelined execution of these steps, there are no race conditions between two *job_submission* packets that do not fix a pointer, between two *task_request* packets, or between *task_request* and *job_submission* packets. These packets do not have a race condition because they access the pointers in (1) serially, one at a time. Consequently, race conditions can only occur between repair packets and retrieval / submission packets because repair packets update the pointers in (1) and update the repair flags in (3).

4.7.1 Race between repairs and a *job_submission*.

If two *job_submission* packets detect that one of the pointers need fixing, they may recirculate two repair packets to correct it. To avoid this scenario, Draconis uses the flags in (3). Because packets are processed serially, one of the *job_submission* packets will reach (3) first, set the flag, and recirculate a repair packet (4). The second *job_submission* packet sees that the flag is already set and does not recirculate a repair packet. When the pipeline receives a repair packet, it fixes one of the pointers in (1) and clears the flag in (3).

4.7.2 Race between repairs and *task_requests*.

If a *task-_request* detects that the repair flag for the `retrieve_ptr` is set (i.e. it entered the pipeline before a repair packet), Draconis returns a no-op to the executor. There is no race scenario if a *task_request* packet enters the pipeline right after a repair packet for the `retrieve_ptr`. The repair packet will fix the `retrieve_ptr` and clear the flag before the *task-_request* reaches these stages.

Task request packets are not affected by a repair packet that fixes the `add_ptr`, as this only occurs when the queue is full and task retrieval can proceed as usual.

4.8 Putting it together: cFCFS Scheduling

With the building blocks outlined in §4, we can build a first-come-first-served (FCFS) scheduler on programmable switches. The client creates *job_submission* packets consisting of a set of tasks and sends them to the switch. The switch enqueues each task into the circular queue. Executors pull tasks from the switch when they are idle by sending *task_request* packets. The switch retrieves the task at the head of the circular queue and sends a *task_assignment* packet back to the executor, with the `TASK_INFO` required to execute the task.

5 Constraint-based Scheduling

While schedulers such as R2P2 [17] support only the FCFS scheduling policy, modern workloads require the use of more complex policies, such as *constraint-based* policies, where individual tasks have special requirements. An example is data locality-aware scheduling. In this policy, the scheduler *prefers* to place a task on the node that contains the data it accesses. Another policy is resource-aware scheduling where tasks *require* specific resources for execution (e.g., GPU) and the scheduler assigns tasks only to nodes with these resources.

5.1 Task Swapping

In order to meet the needs of constraint-based scheduling, we may need to retrieve tasks other than the one at the head of the queue. Draconis first retrieves the task at the head of the queue and inspects it. If the task cannot be scheduled on this executor due to the policy, we must return the task to the queue and fetch the next one from it. This requires a novel task swapping technique in which we swap the task we just popped from the queue with the task at the head of the queue without changes to the queue's `add` and `retrieve` pointers.

To swap tasks, the scheduler creates a special *swap_task* packet. The packet has the task information; `SWAP_INDX`, the index of the next entry to examine in the queue; `EXEC_PROPS`, the executor's properties; and `pkt_retrieve_ptr` containing the current retrieve pointer value. The scheduler populates and recirculates this *swap_task* packet.

When the scheduler receives the *swap_task* packet again, it exchanges the task (`TASK_INFO`) in the packet with the task at the `SWAP_INDX` position in the queue. The packet does not increment the `retrieve_ptr`. If the new task satisfies the scheduling policy, it is sent to the executor. Otherwise, the scheduler repeats the swap logic by incrementing `SWAP_IDX` within the packet and recirculating it.

To avoid complex concurrency conflicts, the *swap_task* packet also contains the retrieve pointer value stored as `pkt_retrieve_ptr`. If the scheduler receives a *swap_task* packet with a `pkt_retrieve_ptr` value that is lower than the current `retrieve_ptr`, then the scheduler will ignore the packet's `SWAP_INDX` value and swap its task with the task at the head of the queue. This is done to avoid scenarios where the task within the packet is swapped into a location which has already been passed over by the `retrieve_ptr` and is lost.

The scheduler recirculates the *swap_task* packet for a bounded number of times (specified by the policy) or until it reaches the end of the queue. If the `SWAP_INDX` in the packet is larger than `add_ptr`, indicating that no task in the queue can run on this executor, the scheduler treats the *swap_task* packet as a *job_submission* packet, inserts the task following the logic in §4, and sends a no-op task to the executor.

We note that the task swap operation maintains the relative order of tasks in a queue. That is because a task in the *swap_task* packet is swapped with the next task within the queue. Furthermore, to avoid starvation, a scheduling policy can specify how many times a task can be swapped.

We will now discuss how task swapping can be used to support two constraint-based scheduling policies: a resource-constrained policy in which a client specifies a hard requirement of the resources needed to execute a task, and data locality-aware scheduling in which a client indicates a preference to execute a task on nodes that have the input data (i.e., soft requirement).

5.2 Resource Constraint-Aware Scheduling

Tasks may require specific resources, such as a GPU or large memory. In this section, we use task swapping to support a policy which takes hard requirements into account. We explore how to build a policy with soft constraints in the following section. Similar to MapReduce [37] and Spark [18], Draconis supports binary task constraints, i.e., the task either needs a resource or not. Tasks may have multiple constraints.

Job Submission. To support resource constraints, the task properties field (TPROPS) in the *job_submission* packets will now be used to indicate the resources required by the task. We use the field as a bitmap where each bit indicates if a certain resource is required by the task. A client sets the appropriate flags in the field to specify the resources required by the task. The scheduler uses the same logic described in §4.3 to process the job submission packet.

Task Retrieval. The scheduler aims to assign a task to the first executor that has the requested resources. When an executor sends a *task_request* packet, it specifies the resources that it has in a bitmap called EXEC_RSRC.

When the scheduler receives this *task_request* packet, it retrieves a task from the queue following the process described in §4.6, i.e., it increments `retrieve_ptr` and fetches a task from the queue. If the queue is empty, a no-op task is sent to the executor. However, if a valid task is retrieved, the scheduler will compare the task requested resources in the TPROPS field to the executor's resources (EXEC_RSRC). If the executor has the resources required to run the task, the scheduler sends the task to the executor.

When the retrieved task cannot be executed by this executor, Draconis reinserts the task into the task queue and retrieves another one, via task swapping. If the queue only has tasks requiring resources that the executor does not possess, a no-op is sent back to the executor.

5.3 Locality-Aware Scheduling

The data locality-aware scheduling policy gives preference for scheduling a task on a node that has the input data. Similar to Spark [18], Draconis supports multiple levels of locality. Each task is tagged with the IDs of nodes that hold the task's data. The scheduler tries to place the task on one of those nodes. After a few attempts, if all the data-local nodes for a task are not free, the scheduler tries to place the task on a node in the same rack as one of it's data-local nodes. If none of the nodes in the same rack are available, after a few attempts, the task is placed on any available node.

Similar to Spark's [18] design, Draconis maintains a `skip_counter` that counts the number of times a task has been examined and skipped over, when looking for a suitable task for scheduling. This additional field is stored in the circular task queue (§4.1). This policy has two configuration parameters: `rack_start_limit` and `global_start_limit`. If the `skip_counter` is larger than `rack_start_limit`, nodes on the same rack as data-local nodes will be considered for scheduling the task. If the `skip_counter` is larger than `global_start_limit`, the task will be scheduled on the next available executor regardless of data locality.

Job submission. Within this policy, the task property field (TPROPS) in the *job_submission* packets is used to hold the node ID of the nodes storing the input data for a task.

Task retrieval. When sending a *task_request* packet, executors include the ID of the node they are running on. When processing this packet, the scheduler retrieves a task from the queue. The scheduler then examines the task property field (TPROPS). If the node the executor is running on is also the task's data-local node, the task is scheduled on the executor.

If the executor is not on a data-local node for the task, the `skip_counter` is incremented and examined. Depending on its value, the following scenarios may arise:

- If the value is less than the `rack_start_limit`, the task is swapped with the next task in the queue.
- If the value lies between the `rack_start_limit` and the `global_start_limit`, the task is scheduled if the executor's node is on the same rack as one of the task's data-local nodes. Otherwise, the task is swapped.
- Finally, if the value is greater than the `global_start_limit`, the task is scheduled on this executor regardless of data-locality.

We note that the number of times a task is recirculated is bounded by the `global_start_limit`. Using the configurable `rack_start_limit` and `global_start_limit`, the scheduling of certain tasks can be selectively delayed in hope that a data-local node will be available to execute it.

6 Classes-of-Service Scheduling

In classes-of-service scheduling, tasks are placed in categories, with each category having different scheduling requirements. To support such policies, we deploy multiple queues on a switch. The circular queue described within §4 consumes a nominal amount of switch resources. As a result, multiple queues can be housed within a single switch and can operate independently. We demonstrate how we use this technique to build task-priority aware scheduling policy.

6.1 Priority-Based Scheduling

Priority-based scheduling is a common scheduling approach in modern schedulers [13, 18, 19, 21]. Unlike schedulers within current frameworks (Hadoop [37] and Spark [18]) that support priority-based scheduling at the job level, Draconis offers priority-based scheduling at the task level, meaning tasks in the same job may have different priorities. Tasks within the same priority level are executed in FCFS order. Higher priority levels have lower priority numbers i.e priority level 1 is the highest priority.

Job Submission. We deploy a separate queue for each priority level. We use the task properties field (TPROPS) in a *job_submission* packets to hold the task's priority level. When the scheduler receives a *job_submission* packet, we insert each task into the queue matching its priority level.

Task Retrieval. When a scheduler receives a *task_request* packet from an executor, it returns the first available task of the highest priority level. To do so, each *task_request* packet has a retrieve priority field (RTRV_PRIO). When an executor submits a *task_request* packet, it sets RTRV_PRIO to 1, the highest priority level supported in the system.

The scheduler will retrieve a task from the FCFS queue corresponding to the priority level specified in the RTRV_PRIO field. If the retrieved task is a valid task, the scheduler forwards the task to the executor. If the retrieved task is an invalid task, indicating that the selected queue is empty, the scheduler will increment the RTRV_PRIO field and recirculate the packet. Incrementing RTRV_PRIO makes the scheduler retrieve a task from a lower priority queue. If RTRV_PRIO becomes larger than the number of priority levels in the system, indicating that there are no tasks at any priority level, the scheduler sends a no-op packet to the executor.

In the worst case, Draconis recirculates a *task_request* packet up to the number of priority levels, which are typically only a handful. In §8.7, we show that this adds negligible overhead. We also note that in newer switches with enough match-action stages, recirculation can be completely avoided by hosting the queues for each priority level on separate stages.

7 Implementation

Our implementation of Draconis supports all the scheduling policies discussed in §4.8, §5 and §6. We have implemented the scheduling logic on a Barefoot Tofino [22] switch using the P4-14 [38, 39] programming language in ~1500 lines of code per policy.

The system uses DPDK-based executors and clients which are implemented in ~1000 lines of code. The clients are designed to submit jobs with configurable sizes, task durations, and interarrival times.

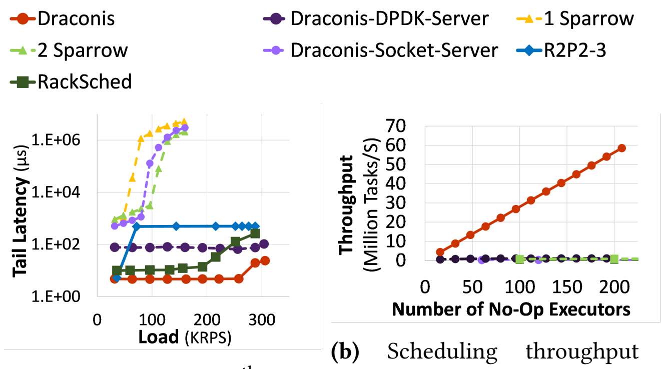
Our switch is one of the earlier P4 programmable switch models (§8) and has limited resources. For instance, our task queue size is 164K and we can only support up to 4 priority levels with the priority-aware scheduling policy. However, newer Barefoot Tofino switches have a significantly larger memory and number of stages [26] which will enable Draconis to overcome these limitations. We estimate that Draconis can have a queue size of 1 million tasks and up to 12 priority levels on Tofino 2 switches [26]. We anticipate newer switches will increase these capabilities further.

8 Evaluation

We compare the performance of Draconis against state-of-the-art centralized, decentralized, and network-accelerated schedulers using synthetic and real-world workloads.

Testbed. We perform all our experiments on a 13-node cluster. Each node has 48 GB of RAM, an Intel Xeon Silver 10-core CPU with hyperthreading, and a 100 Gbps Mellanox NIC. We use 10 nodes as worker nodes running 16 executors each (for a total of 160 executors) unless mentioned otherwise. The remaining nodes are used as schedulers for software-based scheduling systems.

The nodes are connected by an Edgecore Wedge-100BF-32X switch with a Barefoot Tofino ASIC [22].



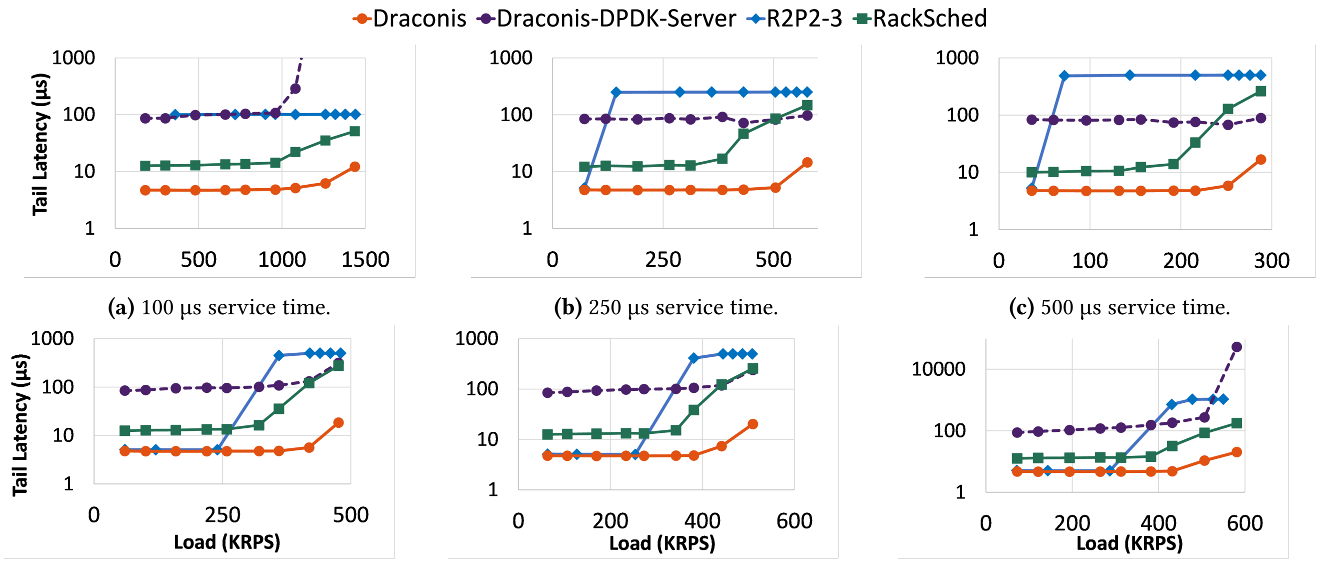
(a) Throughput vs 99th Percentile latency with 500 μs tasks. Y-axis is in log scale.

(b) Scheduling throughput when running a no-op workload

Figure 5. Comparison of all scheduling alternatives

Schedulers. We compare the following server-based and network-accelerated scheduling systems:

- **Draconis.** We use our Draconis implementations on P4. Unless otherwise specified, we use Draconis with a FCFS scheduling policy.
- **Draconis-Socket-Server.** An optimized centralized scheduler following the Draconis scheduling protocol implemented using C++ and Linux sockets.
- **Draconis-DPDK-Server.** An optimized centralized scheduler following the Draconis scheduling protocol, implemented using DPDK [29].
- **R2P2.** We use R2P2's P4 implementation [17]. Unless otherwise specified, we use the configuration used by the R2P2 authors and set the executor queue size to 3 tasks per executor (one task is processed while up to 2 tasks can be queued). We evaluate the impact of varying the executor queue size in §8.3.
- **RackSched.** RackSched [12] is a two-layer scheduler consisting of a P4-based inter-node scheduling component and an intra-node scheduling component. Racksched's dependencies, Shinjuku [10] and Dune [40] , do not run on newer Linux kernels [41] and only support specific Intel NICs. We use RackSched's P4 scheduler and ported it's intra-node scheduler onto DPDK [29] instead of using Shinjuku over Dune. We use the JSQ + cFCFS scheduling policy as recommended by the RackSched authors [12] for light-tailed workloads.
- **Sparrow.** Sparrow is the state-of-the-art distributed scheduler. Our evaluation of the open-source Sparrow implementation [42] showed that it is inefficient as it uses Java and Thrift RPCs. We re-implemented Sparrow in C++ using sockets, achieving 25× higher throughput and 2× lower latency than the original Java implementation. In our evaluation, we use our C++ implementation of Sparrow.



(a) 100 µs service time.

(b) 250 µs service time.

(c) 500 µs service time.

(d) Bimodal workload.

(e) Trimodal workload.

(f) Exponential workload.

Figure 6. Throughput vs 99th percentile of the scheduling delay on synthetic workloads. Note the log scale y axis.

Other Schedulers. We have experimented with the Spark native scheduler, but it could not run sub-second tasks. We have also experimented with Firmament [13], whose open-source implementation could not handle microsecond-scale workloads as well.

Workloads. We use a synthetic workload suite as well as the Google cluster traces [43] to compare Draconis to the alternatives. Our synthetic suite includes workloads with fixed execution time of 100 µs, 250 µs, 500 µs; bimodal (50% 100 µs and 50% 500 µs) and trimodal (33.3% 100 µs, 33.3% 250 µs, and 33.3% 500 µs) workloads. It also contains a synthetic workload with an exponential distribution of execution times, with a mean of 250 µs.

In order to assess performance with a real workload, we use the Google cluster traces [43] in our evaluation. The traces have been accelerated to create two separate versions with mean task execution times of 500 µs and 5 ms. The traces contain task priority information, which are used to evaluate our priority-aware policy. In all our experiments, we report the average of 10 runs. The standard deviation in all our experiments was under 5%.

8.1 Scheduling Latency

Figure 5a plots the 99th percentile of the scheduling delay against increasing system load for Draconis and the alternatives, when running a synthetic workload with 500 µs tasks. We have deployed both a single (1 Sparrow) and two Sparrow schedulers (2 Sparrow) on our cluster for this experiment.

Figure 5a shows that Draconis has significantly lower tail latencies compared to other systems at 4.7 µs, *3×*, *20×*, *120×* and *200× lower* than that of RackSched, Draconis-DPDK-Server, R2P2 and Sparrow respectively. The tail latency increases when the load is above 250k tasks per second (tps), as the cluster utilization is over 90% and tasks experience queuing delays. Nevertheless, even at high cluster utilization Draconis achieves 30× lower latency than the closest alternative.

The figure also demonstrates that systems that use POSIX sockets cannot support more than 160k tps and experience 200× higher tail latency compared to Draconis. Sparrow is outperformed even by Draconis-Socket-Server which achieves 1.7× lower latency than a single Sparrow scheduler and comparable performance to the dual Sparrow scheduler deployment. This is because Sparrow experiences higher latencies due to its additional probing overheads. Sparrow also does not achieve ideal task placement as it probes only a fraction of the cluster for each task. Sparrow and Draconis-Socket-Server could not run workloads with lower execution times and thus, we omit their results in the following sections.

Figure 6 shows the 99th percentile of the scheduling delay plotted against increasing system load, with the entire suite of synthetic benchmarks. Draconis consistently achieves tail latencies of 4.7–20 µs, significantly lower than RackSched, R2P2 and Draconis-DPDK-Server.

R2P2 always selects the node with the shortest queue. As demonstrated by the authors of RackSched [12], this leads to *herding* i.e. batches of tasks are sent to the executor with the shortest queue before the queue length is updated. While R2P2 keeps pace with Draconis at low loads, as the load increases R2P2 suffers from node-level blocking at the executors. For instance, with uniform workloads, its tail latency is

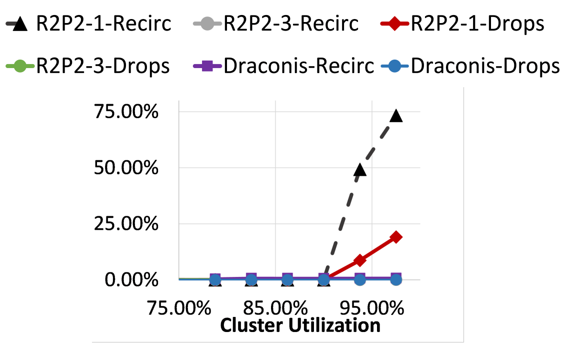
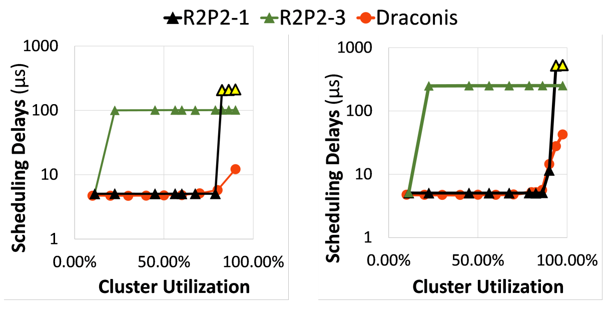


Figure 7. Examining Task Drops with the 250 µs workload



(a) 100 µs tasks. **(b)** 250 µs tasks

Figure 8. Cluster Utilization vs 99th Percentile scheduling delays for R2P2 configurations. Yellow triangles indicate runs with dropped tasks.

always equal to the task service time. This begins to occur at 30%–40% cluster utilization.

RackSched avoids the problems associated with herding as it uses sampling and the power-of-two choices to select the server with the shortest queue. However, as the load on the cluster increases, sampling is ineffective and leads to node-level blocking. This increases its tail latency under high loads. In addition, RackSched's intra-node scheduling component incurs a small additional overhead (3–4 µs) which impacts its latency even under low cluster utilization.

8.2 Scheduling Throughput

Microsecond-scale workloads on large clusters require the processing of thousands of status reports and making millions of scheduling decisions per second. State-of-the-art sever-based schedulers cannot scale to support these workloads on large clusters, while maintaining low tail latencies. For instance, Firmament, a state-of-the-art centralized scheduler, can only support a cluster of up to 100 nodes when running millisecond-scale workloads [13].

To demonstrate the viability of network-accelerated scheduling for microsecond-scale workloads, we compare the throughput of Draconis against all the server-based scheduling alternatives. To measure the throughput of each scheduling system, we use a synthetic workload composed of no-op tasks, i.e., an executor retrieves the task, immediately drops it, and requests a new task. We vary the number of executors to increase the load on the scheduling system.

Figure 5b shows scheduling throughput with increasing number of no-op executors. Draconis' performance improves linearly with additional executors, achieving a throughput of 58 million scheduling decisions per second with 208 executors, *52× higher* than Draconis-DPDK-Server, the closest alternative. Unfortunately, we could not deploy more than 208 executors on our cluster to stress Draconis. The switch can handle up to 4.7 billion packets per second, indicating that Draconis' peak throughput is significantly higher than the workload our no-op executors can generate. We use simulations to evaluate Draconis' scalability. Our simulations show that Draconis can support clusters of millions of cores when running 500 µs tasks.

We note that server-based schedulers do not scale well. Draconis-DPDK-Server has the highest throughput of 1.1 Mtps while Sparrow's throughput is the lowest at 500k and 900k tps for single and dual Sparrow schedulers. We construe that supporting microsecond-scale workloads on large clusters at low tail latencies will require hardware-acceleration, such as the use of P4 programmable switches.

8.3 The Effect of JBSQ size on R2P2

R2P2 can be run with varying JBSQ executor queue size. The executor queue size impacts the tail latency [17]. However, in practice, the process is not as straightforward. Figures 8a and 8b compare the scheduling tail latencies of Draconis and R2P2 with two configurations of the executor queue size. Within R2P2-1, each executor has no queues and gets 1 task to execute at a time. Within R2P2-3, each executor can hold 3 tasks; 1 task is executed and 3 tasks can be queued. We use workloads with 100 µs and 250 µs tasks respectively. We note that R2P2-1 has no queue in the system to absorb task submission bursts, which leads to it dropping tasks. This is confirmed by our experiments below.

Under low cluster utilization R2P2-1 achieves a tail latency comparable to Draconis. As the load increases, R2P2-1 drops an increasing number of tasks. Cluster loads with dropped tasks are highlighted using yellow markers in both figures. For instance, in Figure 8a at a cluster load of 82%, R2P2-1 drops 5% of tasks at the switch. In Figure 8b, R2P2-1 drops 9% of tasks at a cluster load of 93%. A client will timeout and resubmit these tasks, causing a spike in tail latency in the figures. Typically, clients use timeout values ranging between 5–10× of the task execution time. In our experiments, we have set the client timeout to 2× the task execution time.

Figure 7 provides further insight into the cause of these dropped tasks. R2P2 heavily relies on packet recirculation to find available executors (§2.2). If no executor is available, a request is continuously recirculated within the switch. With

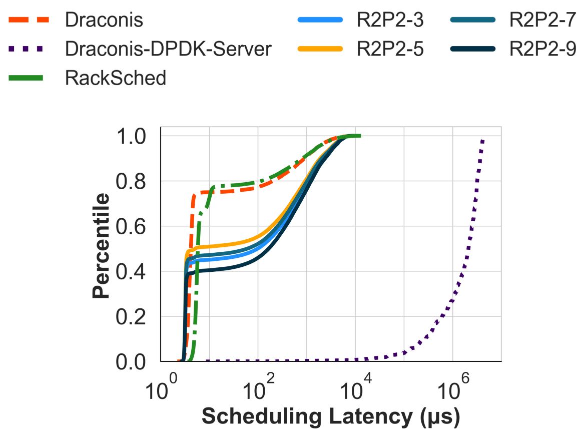


Figure 9. Scheduling Latency CDF of different alternatives when running a real workload. X-axes are in log scale.

increasing cluster utilization, R2P2 suffers from a larger number of recirculations. Figure 7 shows the percentage of recirculated packets over the total packets processed by R2P2-1 with increasing cluster load (R2P2-1-Recirc), when running 250 µs tasks. At a load of 93%, almost 50% of all processed packets are recirculations. This number increases to 75% at a load of 97%. While the switch possesses a large packet bandwidth, its recirculation bandwidth is far more limited, thus resulting in many tasks being dropped while being recirculated. This makes R2P2-1 unsuitable for handling real workloads with bursts.

The solution provided by the authors of R2P2 to address this is to have larger executor queue sizes. Indeed, increasing the queue size to 3 on each executor brings down the number of recirculations and dropped tasks to zero, as shown in Figure 7. However, R2P2-3 experiences higher tail latencies. Figures 8a, 8b show that with a cluster utilization of 30%–40%, its tail latency is equivalent to the task execution times of 100 µs and 250 µs respectively. This is because of node-level blocking at the executor queues. For the rest of our evaluation we use R2P2-3, the configuration used by the R2P2 authors [17] that does not drop tasks.

Figure 7 also shows that Draconis can handle high cluster utilization levels without dropping any tasks. This is because, Draconis uses recirculations far more sparingly. Figure 7 shows that the percentage of recirculated packets in Draconis ranges from 0.02% to 0.05%.

8.4 Performance with a Real Workload

We evaluate scheduling latencies using the Google cluster traces [43], which include information about tasks running on a 12,500-node cluster at Google for over a month. To generate a trace that we can run on our 12-node cluster, we followed an approach similar to that of Firmament [13]. We took a uniform sample of the trace and accelerated it to run on our cluster in 3 minutes. The resulting trace had a mean task duration of 500 µs. The google trace is bursty i.e. it may submit hundreds of tasks at once. Each task continually performs integer arithmetic operations for the task duration. We evaluate this workload by running 16 executors each on 10 worker nodes (160 executors total).

Figure 9 is a CDF of the scheduling delay using the alternative systems. We use four JBSQ queue sizes while running this workload with R2P2. A JBSQ size of 1 (R2P2-1) caused 6.3% of tasks to be dropped, hence we omit it from the figure.

Figure 9 shows that the median scheduling delay of Draconis is around 4.18 µs. R2P2 with a JBSQ size of 5 (R2P2-5) is the best performing R2P2 variant with a median scheduling delay of 5.2 µs while the other variants have median latencies varying from 60–160 µs. RackSched has a median scheduling delay of 5.83 µs. This shows that Draconis' median scheduling delay is 24% and 39% lower than R2P2-5 and RackSched respectively. The tail latencies for Draconis at the 95^{th} and 99^{th} percentile are better than R2P2-5 by 200% and 20% respectively and are similar to RackSched's tail latencies. The increase in scheduling delay and long tails for all alternatives is due to the bursty nature of the google trace, which results in queuing.

We note that that increasing JBSQ queue size does not translate to better performance, as R2P2-5 performs better than R2P2-7 and R2P2-9 (Figure 9). On the other hand R2P2-3 queue size is not large enough, causing recirculation at the switch and affecting performance.

The median scheduling delay for Draconis-DPDK-Server is orders of magnitude higher than all network-accelerated schedulers at around 2.1 seconds. This is because server-based schedulers cannot handle the high throughput requirements of microsecond-scale workloads (§ 8.2).

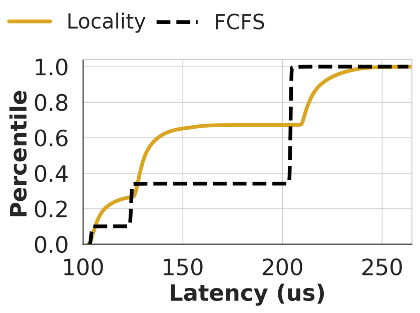


Figure 10. Locality-aware scheduling vs FCFS.

8.5 Constraint-based Scheduling Policies

Locality-Aware Scheduling. To evaluate our locality-aware scheduling policy, we designed an experiment that emulates a multi-rack deployment. We divided our worker nodes into 3 racks. Each node runs 16 executors. We set the intra-rack and inter-rack storage access times to 20 µs and 100 µs [44] respectively.

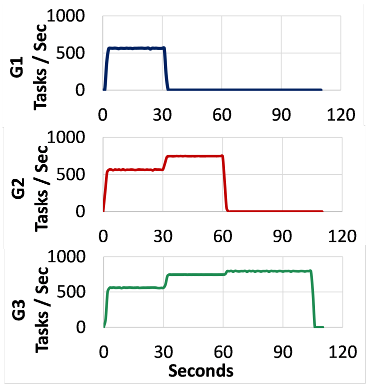


Figure 11. System throughput with resource constraints.

We run a CPU-intensive synthetic locality-aware workload consisting of 100 µs tasks. The processed data is not replicated and is evenly partitioned across the nodes. Thus, each task has it's data local to one node in the cluster.

We run this workload with different configurations of `rack` and `global` start limits. Figure 10 shows the CDF of the scheduling delay with a `rack_start_limit` of 3 and a `global_start_limit` of 9. This configuration schedules 27.66% of the workload on their preferred nodes, 38.82% of tasks on the same rack as their preferred node, and the rest on another rack. In comparison, Draconis-FCFS only places 10.03% of tasks on their preferred node, 24.05% on the same rack, and the remaining 65.94% on a different rack. We experimented with other values for these limits and noticed that at least 49% of tasks are scheduled on the target node or rack in all configurations.

Figure 10 shows the CDF of the total end-to-end delays experienced by tasks scheduled using Draconis-Locality against Draconis-FCFS. We note that Draconis-FCFS has a median latency of 203.87 µs while Draconis-Locality has a median latency of 131.35 µs. Draconis-Locality performs 2× better at the 66th percentile, after which Draconis-FCFS achieves better latencies due to incurring the same locality overhead without delaying the scheduling of tasks.

Resource-Aware Scheduling. To demonstrate the effectiveness of resource-constraint aware scheduling, we design the following experiment. We assume that the cluster has three types of resources: A, B, and C. These can represent different resources (e.g., GPU, large memory, hardware accelerators). We divide the cluster nodes into three groups: G1 has resource A, G2 has resources A and B, and G3 has resources A, B, and C. Tasks can specify the resources they need. For simplicity, we design a synthetic benchmark where each task requests only one of the resources A, B, or C.

The experiment runs for 90 seconds. In the first 30 seconds, all submitted tasks require resource A, available on all nodes. In the next 30 seconds, all tasks require resource B, available on G2 and G3 nodes. In the last 30 seconds, all tasks require resource C, only available on G3 nodes.

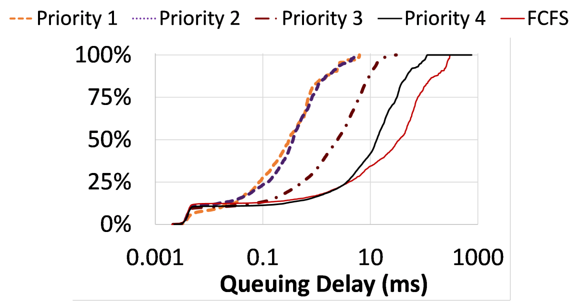


Figure 12. CDF of queuing delays across different priority levels. The x-axis is in log scale.

Figure 11 shows the average throughput of a node from each one of the three node groups. In the first 30 seconds, all nodes in all groups are busy, as all nodes have the requested resource A. In the next 30 seconds, only the nodes in G2 and G3 are running tasks. In the last 30 seconds, only G3 nodes are running tasks. We note that G3 nodes are overloaded. Thus, although the last task is submitted at the 90 seconds mark, the execution only finishes at the 110 seconds mark.

8.6 Class-of-Service based Scheduling Policies

To demonstrate priority-based scheduling, we use an accelerated Google trace with a mean task execution time of 5 ms. We have increased the sampling rate to place higher load on the cluster, thereby increasing the queuing delays. The Google traces [43] have 12 levels of priority, while our implementation has four. We map every three levels of Google priorities to one priority level in Draconis. The resulting workload has 1.2%, 1.7%, 64.6%, and 32.2% of tasks at priority levels 1, 2, 3, and 4, respectively. Tasks at different priority levels experience different queueing delays, with higher priority tasks experiencing shorter queueing delays. Figure 12 shows the queueing delays of tasks at different priority levels. Tasks with priority levels 1, 2, 3 and 4 have median queueing delays of 1.4 ms, 2.9 ms, 13.3 ms, and 53.5 ms, respectively. The same workload run with a priority unaware FCFS policy (FCFS in Figure 12) has a median queuing delay of 39.5 ms. Priority 1 (highest priority) tasks are only queued when there are no free executors to run them, leading to the lowest queueing delay.

8.7 Packet Recirculation Overheads

Heavy packet recirculation can impact the switch throughput (§8.3). In this section, we evaluate the recirculation overhead for each scheduling policy in Draconis.

FCFS Policy. In the FCFS scheduling policy, recirculation is only used to fix pointers when the queue is empty / full and to handle job submissions with multiple tasks. In our experiments, recirculated packets make up only 0.02 − 0.05% of all processed packets by the switch even at high cluster loads (Figure 7). This small percentage of recirculation does not impact system throughput.

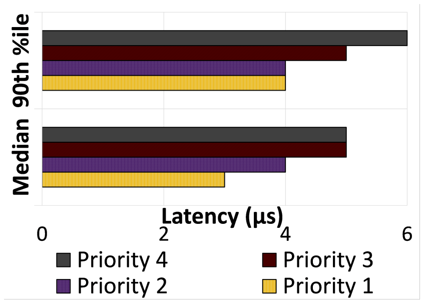


Figure 13. `Get_task()` delays across different priority levels.

Constraint-based policies. The amount of packet recirculation within constraint-based policies is governed by user configurable parameters, such as the `rack` and `global` limits. For the configuration in Figure 10, the total percentage of recirculated packets is less than 1%. This small percentage of recirculation does not impact system throughput.

Priority-Based Policy. Draconis' priority-based scheduling currently uses packet recirculation to check the task queues at different priority levels. Packet recirculation typically takes less than a microsecond. Figure 13 shows the latency of the `get_task()` step from Figure 2. The median and 90th percentile latencies between priority levels differ by just 1–2 µs at most. Thus, the latency overhead imposed by packet recirculation is negligible. Additionally, we did not observe any impact on scheduling throughput up to 58M tasks / second, the peak load our cluster is able to generate (Figure 5b).

We note that, due to the limited number of match-action stages in our switch, we place all the task queues in the same set of stages and use recirculation to check multiple priority queues. Newer programmable switches have double the number of stages as ours and can house each task queue in separate stages, completely bypassing the need for packet recirculation.

9 Additional Related Work

Hybrid Scheduling. Hawk [45] and Mercury [46] propose a hybrid paradigm involving centralized scheduling for long-running jobs and decentralized scheduling for low-latency jobs. However, they suffer the same drawbacks as their decentralized counterparts when scheduling microsecond-scale tasks.

Streaming Systems. Numerous systems [47–49] have been designed to tackle sub-second tasks in the streaming environment. However, they neither target dynamic scheduling nor workloads in the microsecond-scale.

Network-Accelerated Systems. Many recent projects have used programmable switches to accelerate consensus protocols [33, 34, 50, 51], implement in-network caching [52], accelerate DNN training and inferencing [53], and support in-network aggregation operations [54]. These are orthogonal to Draconis. JumpGate [55] proposed offloading some data analytics functions to the switch but did not investigate supporting in-network scheduling.

Our previous workshop paper [56] examines the feasibility of building an in-switch centralized scheduler. However, the paper does not present a complete system design that supports multiple scheduling policies nor evaluates the proposed approach against the state-of-the-art.

Low-Latency Optimizations. Several projects have explored operating system and network stack optimizations for low latency workloads, including kernel-bypass techniques [29, 57, 58] and efficient core reallocation mechanisms [11, 59]. These efforts are orthogonal to ours as we design a scheduler to support microsecond-scale workloads.

10 Concluding Remarks

We present Draconis, a centralized in-network scheduler that can support microsecond-scale workloads on large clusters. Draconis adopts a fundamentally different design approach compared to the state-of-the-art by designing a switch compatible task queue and using a pull-based scheduling model. Our evaluation shows that Draconis can reduce scheduling overheads by an order of magnitude and achieve significantly higher throughput compared to state-of-the-art schedulers. Draconis demonstrates that despite their strict programming and memory model, modern programmable switches can be leveraged to implement complex data structures and scheduling policies. The Draconis source code is available on Github [30].

Acknowledgments

We thank Ibrahim Kettaneh for his contributions to an earlier version of this work. We would like to thank Mina Tahmasbi Arashloo for her feedback on the system design and Bernard Wong, Khuzaima Daudjee and Robin Grosman for their feedback on earlier versions of this paper. We thank the anonymous EuroSys reviewers and our shepherd, Daehyeok Kim, for their detailed feedback which has enhanced this paper.

References

[1] D. Meisner, C. M. Sadler, L. A. Barroso, W. Weber, and T. F. Wenisch. Power management of online data-intensive services. *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 319–330, 2011.

[2] Xinhui Tian, Rui Han, Lei Wang, Gang Lu, and Jianfeng Zhan. Latency critical big data computing in finance. *The Journal of Finance and Data Science*, 1(1):33–41, 2015.

[3] Ciamac Moallemi and Mehmet Saglam. OR forum—the cost of latency in high-frequency trading. *Operations Research*, 61(5):1070–1086, 2013.

[4] Stephen F. Elston and Melinda J. Wilson. Big data and smart trading. *https://www.risktechforum.com/media/download/61681/download.*

[5] Boming Huang, Yuxiang Huan, Li Da Xu, Lirong Zheng, and Zhuo Zou. Automated trading systems statistical and machine learning methods and hardware implementation: a survey. *Enterprise Information Systems*, 13(1):132–144, 2019.

[6] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[7] Ramana Rao Kompella, Kirill Levchenko, Alex C. Snoeren, and George Varghese. Every microsecond counts: Tracking fine-grain latencies with a lossy difference aggregator. *SIGCOMM Comput. Commun. Rev.*, 39(4):255–266, aug 2009.

[8] Kay Ousterhout, Aurojit Panda, Joshua Rosen, et al. The case for tiny tasks in compute clusters. *Proceedings of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[9] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 599–616. USENIX Association, November 2020.

[10] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for usecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[11] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, pages 361–377, 2019.

[12] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. *the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[13] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[14] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, Boston, MA, July 2018. USENIX Association.

[15] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013.

[16] W. Chen, A. Pi, S. Wang, and X. Zhou. Characterizing scheduling delay for low-latency data analytics workloads. *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 630–639, 2018.

[17] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2p2: Making rpcs first-class datacenter citizens. *2019 USENIX Annual Technical Conference (ATC 19)*, 2, 2019.

[18] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, 10:10, 2010.

[19] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: distributed low latency scheduling. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84, 2013.

[20] Xiaoqi Ren, Ganesh Ananthanarayanan, Adam Wierman, and Minlan Yu. Hopper: Decentralized speculation-aware cluster scheduling at scale. *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 379–392, 2015.

[21] Eric Boutin, Jaliya Ekanayake, Wei Lin, et al. Apollo: Scalable and coordinated scheduling for cloud-scale computing. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, 2014.

[22] Tofino world's fastest p4-programmable ethernet switch asics. Retrieved from https://www.barefootnetworks.com/products/brief-tofino/.

[23] Mark Van der Boor, Sem C. Borst, Johan S. H. Van Leeuwaarden, and Debankur Mukherjee. Scalable load balancing in networked systems: A survey of recent advances. *SIAM Review*, 64(3):554–622, 2022.

[24] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 466–481, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Operations Research*, 60(5):1249–1257, 2012.

[26] Tofino-2 second-generation of world's fastest p4-programmable ethernet switch asics. Retrieved from https://www.barefootnetworks.com/products/brief-tofino-2/.

[27] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[28] Panagiotis D. Diamantoulakis, Vasileios M. Kapinas, and George K. Karagiannidis. Big data analytics for dynamic energy management in smart grids. *Big Data Res.*, 2(3):94–101, 2015.

[29] Dominik Scholz. A look at intel's dataplane development kit. 2014.

[30] GitHub - UWASL/Draconis: Draconis: Network-Accelerated Scheduling for Microsecond-Scale Workloads — github.com. https://github.com/UWASL/Draconis. [Accessed 16-02-2024].

[31] Xin Zhe Khooi, Levente Csikor, Jialin Li, and Dinil Mon Divakaran. In-network applications: Beyond single switch pipelines. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 1–8, 2021.

[32] Mellanox connectx 6 vpi product sheet. https://support.mellanox.com/s/productdetails/a2v50000000p8ReAAI/-connectx6-card.

[33] Samer Al-Kiswany, Suli Yang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Nice: Network-integrated cluster-efficient storage. *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 29–40, 2017.

[34] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say no to paxos overhead: Replacing consensus with network ordering. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483, 2016.

[35] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be fast, cheap and in control with SwitchKV. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, Santa Clara, CA, March 2016. USENIX Association.

[36] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[37] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 137–150, 2004.

[38] Pat Bosshart, Dan Daly, Glen Gibb, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[39] P4. Retrieved from https://p4.org/.

[40] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In Chandu Thekkath and Amin Vahdat,

editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 335–348. USENIX Association, 2012.

[41] Kern build problem· issue #25· project-dune/dune. 2023. https://github.com/project-dune/dune/issues/25.

[42] Sparrow git repository. 2013. Retrieved 2023 from https://github.com/radlab/sparrow.

[43] John Wilkes. Google clusterdata 2011 traces. GitHub. Retrieved from https://github.com/google/cluster-data.

[44] Diana Andreea Popescu. Technical report - latency-driven performance in data centres. *Doctoral dissertation, University of Cambridge*, 2019.

[45] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. *USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, 2015.

[46] Konstantinos Karanasos, Sriram Rao, Carlo Curino, et al. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. *USENIX Annual Technical Conference (USENIX ATC 15)*, pages 485–497, 2015.

[47] Boduo Li, Yanlei Diao, and Prashant Shenoy. Supporting scalable analytics with latency constraints. *Proc. VLDB Endow*, 8(11):1166–1177, 2015.

[48] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438, 2013.

[49] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 374–389, New York, NY, USA, 2017. Association for Computing Machinery.

[50] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–7, 2015.

[51] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. Flair: Accelerating reads with consistency-aware network routing. *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 723–737, 2020.

[52] Xin Jin, Xiaozhou Li, Haoyu Zhang, et al. Netcache: Balancing key-value stores with fast in-network caching. *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[53] Dan R. K. Ports and Jacob Nelson. When should the network be the computer? *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 209–215, 2019.

[54] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 150–156, 2017.

[55] Craig Mustard, Fabian Ruffy, Anny Gakhokidze, Ivan Beschastnikh, and Alexandra Fedorova. Jumpgate: In-network processing as a service for data analytics. *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[56] Ibrahim Kettaneh, Sreeharsha Udayashankar, Ashraf Abdel-hadi, Robin Grosman, and Samer Al-Kiswany. Falcon: Low latency, network-accelerated scheduling. In *Proceedings of the 3rd P4 Workshop in Europe*, EuroP4'20, page 7–12, New York, NY, USA, 2020. Association for Computing Machinery.

[57] Ilias Marinos, Robert N. M. Watson, and Mark Handley. Network stack specialization for performance. *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 175–186, 2014.

[58] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[59] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, Renton, WA, April 2022. USENIX Association.

[60] Draconis: Network-Accelerated Scheduling for Micro-Scale Workloads. https://zenodo.org/records/10688915.

A Artifact Appendix

A.1 Abstract

We are releasing Draconis' source code along with our paper. The repository contains the scheduler code and switch-compatible circular queue implementation, the main contributions of our paper. The repository additionally contains the code for workers, clients and Draconis-DPDK-Server used in our evaluation.

A.2 Description & Requirements

A.2.1 How to access. The Draconis code is publicly available on Github [30]. The repository contains a link to our paper and uses the MIT license. The packaged source code is also available via Zenodo [60].

A.2.2 Hardware dependencies. Draconis requires a P4 programmable switch with a P4-14 compiler in order to be deployed. We have used an EdgeCore Wedge switch with a Barefoot Tofino ASIC for our evaluation. Draconis-DPDK-Server as well as workers and clients require machines with a DPDK-compatible NIC.

A.2.3 Software dependencies. Draconis-DPDK-Server and workers / clients require machines with DPDK installed and enabled to function. We have used Intel DPDK v18.11 for our experiments.

Draconis workers and clients require Ubuntu 16.04 or greater on the machine they are deployed on.