A Network Integrated Design for Micro-scale Cloud Services

by

Ashraf Abdel-hadi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

The results of this work are included in two papers. The first, is a position paper at the third P4 Workshop in Europe (EuroP4 '20) [1] and the second is a paper under submission to the Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2023. I am the lead on this project it has been done in collaboration with Ibrahim Kettaneh and Sreeharsha Udayashankar under the supervision of our advisor, Prof. Samer Al-Kiswany. Ibrahim was involved in the basic FIFO Falcon design (Section 4.2). Sreeharsha was involved in the FIFO Falcon design, and the evaluation section.

Abstract

We present Falcon, network-integrated scheduler for micro-scale services. Falcon follows a centralized scheduler design to achieve high scheduling efficiency and leverages modern programmable switches to lower the scheduling latency and increase the scheduling throughput. Falcon supports multiple scheduling policies such as FIFO, and data locality aware policies. Our empirical evaluation shows that Falcon reduces scheduling latency by 120 times and increases the scheduling throughput by 100 times compared to state-of-the-art schedulers.

Acknowledgements

I would like to thank my supervisor Prof. Samer Al-Kiswany for supporting me during my MMath degree. Without his knowledge and guidance, this thesis would not be complete. I could not ask for a better mentor and a role model.

I would like to thank Prof. Ali Mashtizadeh and Prof. Seyed Zahedi for their important feedback.

I would like to thank my friends and collaborators, Sreeharsha Udayashankar and Ibrahim Kettaneh, this thesis would not be possible without their help and support. I would like to extend my gratitude to everyone at Waterloo Advanced Systems Lab (WASL) for keeping me company during my time at UW.

Finally, I would like to thank my Family, and friends for without their love and support I would not reach this milestone. No words can express how much their support and love mean to me. I hope I did them proud.

Table of Contents

List of Figures

1 Introduction

Online data-intensive services [2, 3] such as e-commerce, social networking, object detection [4], algorithmic smart trading [5, 6, 7, 8], and low-latency web services [9] are expected to provide a scalable and interactive services. To meet these requirements, these systems are required to support high throughput and low-tail latency in the range of microseconds [10]. Meeting this requirement is specially challenging for online data-intensive applications which fan out requests to tens of cores. In these applications, the response time is dominated by the slowest task [9].

Traditional data-processing frameworks [11, 12] use centralized scheduler designs. In this design, a central scheduler monitors the cluster nodes and schedules tasks on the next available node. Although this approach makes accurate scheduling decisions using complete cluster information, it cannot support real-time workloads on large clusters because this requires the processing of thousands of status reports and making hundreds of thousands of scheduling decisions per second. For instance, Firmament, a state-of-the-art centralized scheduler, can only support a cluster of up to 100 nodes when running real-time analytics [13], and Spark's scheduler does not support sub-second tasks [12]. Chen et al. [14] characterized the scheduling overhead in modern low-latency data analytics and found that scheduling overheads account for nearly 60% of total execution time. Finally, our evaluation (§7.2) shows that even an optimized centralized scheduler can only support clusters with up to 100 nodes when running latency-sensitive tasks.

To overcome these limitations, a number of data analytics frameworks have explored

a distributed scheduling design [15, 16, 17, 18]. Unfortunately, this approach has two disadvantages. First, it has high overhead because it uses tens of nodes for scheduling; for instance, Sparrow [17] uses 10% of the cluster nodes to run schedulers. Second, this approach leads to suboptimal scheduling decisions leading to higher tail latency. To avoid the overhead of coordinating tens of schedulers, some frameworks use probing, whereby in which each scheduler probes a fraction of cluster nodes and schedules the next task on the first node that becomes free in this subset [17, 18]. Alternatively, Apollo [15] uses a centralized metadata service that periodically monitors the cluster nodes. Unfortunately, the information is often stale and lead to multiple schedulers swarming over the same set of nodes, leading to higher tail latency. Furthermore, probing overhead increases the scheduling overhead.

We present Falcon, a novel scheduler design for low-latency data analytics workloads. Falcon makes accurate scheduling decisions and can support clusters with thousands of nodes. To make precise scheduling decisions Falcon follows a centralized scheduling approach. To improve the scheduling performance and support large clusters Falcon accelerates the scheduler by leveraging modern programmable switches [19, 20].

Modern programmable switches [19, 20, 21] can process billions of packets per second. This means these switches are able to handle scheduling workloads on large clusters while acting as a centralized scheduler. However, using these programmable switches to build complex systems is challenging due to their restrictive pipeline-based programming model, limited computational power, and restrictive memory access model. Simple implementations such as a simple circular queue can be challenging, because their basic methods usually access the queue twice: at the start to determine if the queue has any items or not, and a second time to increase or decrease the queue's size. This is a clear violation of one of the switch's restrictions where a packet can only access a memory unit once.

A critical part of Falcon's design is our novel P4 circular queue, which supports task addition and removal at line-rate speeds. This circular task queue also supports the addition of lists of entries (§4.2). despite these switch's limited capabilities Falcon's is able to support a FIFO scheduling policy as well as a data locality-aware scheduling policy.

Two recent projects explored building a switch-based schedulers for microsecond scale tasks [22, 23]. These two approaches are optimized for single request RPCs and do not readily support data analytics frameworks with micro-batch interface. Furthermore, R2P2 has a high tail latency due to high recirculation overhead and head-of-line blocking. Racksched can not scale beyond tens of servers limited by how many operations it can fit in a Tofino stage, Racksched also relies on a secondary inter-server scheduler to schedule requests to the desired core.

Our prototype evaluation on a cluster with a Barefoot Tofino switch [20] using synthetic and real-world workloads shows the significant benefits of this approach. Our evaluation with real and synthetic benchmark shows that the 99 percentile of Falcon scheduling delay is around 5 µs, at least 120× lower than state of the art scheduler. Falcon can support clusters of millions of executors, orders of magnitude higher than the state-of-the-art centralized and decentralized schedulers. Finally, we demonstrate Falcon's locality scheduling policy.

The rest of the paper is organized as follows. We present the related background in Chapter 2. We discuss Falcon 's design in Chapters 3 and 4, then detail its approach for supporting locality (Chapter 5). We discuss the details of our implementation in Chapter 6. In Chapter 7, we present our evaluation. We survey related work in Chapter 8 and conclude in Chapter 9.

2 Motivation and Background

2.1 Real-Time Analytics

Online data-intensive analytics aim to complete tasks in hundreds of microseconds. Applications such as traffic analysis [24], financial analytics [5], smart-grid monitoring [25, 26], and IoT analytics for defense [27] and agriculture [28] have stringent timing requirements. For instance, lowering the latency of financial analytics by even a millisecond can boost earnings by a hundred million dollars a year [8]. Finally, real-time applications such as rapid object detection [4, 29] and augmented reality applications [9] require analyzing sensors and video data in real time.

Data analytics frameworks need to handle hundreds of thousands or even millions of scheduling decisions per second with tail latency of a few microseconds to be able to run these applications with clusters sized around a few hundreds of nodes.

2.2 Overview of Scheduling Paradigms

Generally, data-processing frameworks use a micro-batch scheduling model [12]. Jobs are submitted with m independent tasks. These tasks can be executed in parallel by executors. A job is complete when all tasks have completed their execution on executors in the cluster.

Centralized Scheduler Design. A single scheduler that utilizes knowledge and status of the cluster to make accurate scheduling decisions. However, this design is unable to support real-time analytics even on small clusters. Firmament [13] is a centralized scheduler that

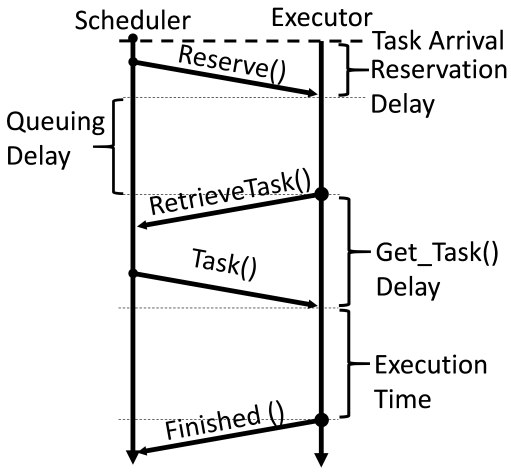


Figure 1: Sparrow's scheduling timeline

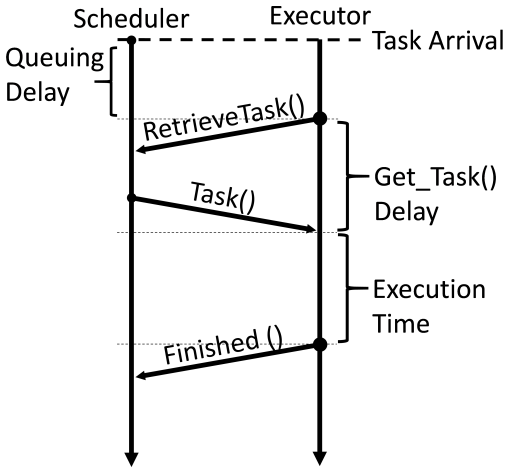


Figure 2: Falcon's scheduling timeline

models the workload into a graph with tasks linked to their assigned executor. Firmament maps these tasks into their appropriate executors using a min-cost max-flow solver. Every time a new job is submitted the calculated graph needs to be updated by running the graph solver to generate a new graph with the new tasks included. Gog et al. [13] report that Firmament is unable to scale to more than 100 nodes (each running 12 executor cores) with real-time workloads, even with their highly optimized solver implementation.

Apache Spark [12] is also a centralized scheduler. Our analysis and that of the authors of Sparrow [17] show that Spark is incapable of running tasks with service time below 1.5 seconds. This is because Spark suffers from infinite queuing with low service times.

Distributed Scheduler Design. Modern distributed schedulers [15, 17, 18] address large-scale clusters support by deploying multiple schedulers. Distributed schedulers make decisions with partial or stale cluster information, that way they avoid the heavy overhead

of communication and coordination between multiple schedulers. For instance, Sparrow [17], a state-of-the-art distributed scheduler, schedules an m tasks job by probing 2m randomly executors(Figure 1). The probes are queued on the executors. When an executor is free, it dequeues a probe and requests a task from the scheduler to execute. Then, the scheduler would forward a task to be executed to the executor. Sparrow uses this technique to offset the partial knowledge of cluster utilization. Hopper [18] adopts a similar technique. Due to probing only a fraction of the nodes scheduling decisions are not optimal, and more overhead is added to the scheduling delay.

Apollo [15] is a distributed scheduler that employs a central resource-monitoring service. Schedulers in the cluster communicate with the resource-monitoring service to update their view of the cluster nodes and their status. The resource-monitoring service does not coordinate the information between schedulers well. Which leads to multiple schedulers forwarding tasks to the same nodes, achieving suboptimal scheduling decisions.

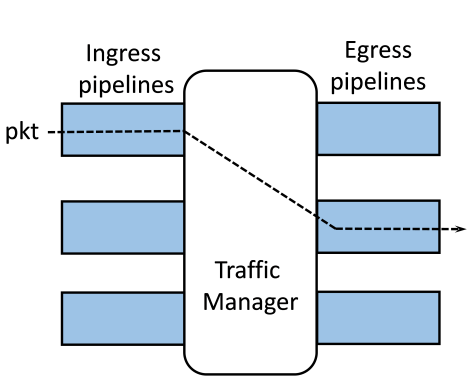
Switch-Based Schedulers. R2P2 [22] implements join-bounded-shortest-queue(JBSQ) in which it aims to schedule the next task on the executor with the shortest queue. To implement this approach in P4, R2P2 maintains an array of counters at the switch to keep track of the queue size of each executor. When a client sends a new RPC call, R2P2 will recirculate the request until it finds an executor with empty queue. If no such executor is found, it will recirculate the request over the array looking for an executor with one task or less, and so forth. This approach suffers from excessive re-circulation, head of line blocking, and leads to lower cluster utilization. Our evaluation shows that when a cluster is 90% utilized, 45% of the processed packets are recircualtion packets. Furthermore, our evaluation shows when the cluster is lightly loaded, R2P2 uses less than 50% of the cluster resource, negatively affecting scheduling tale latency.

Racksched [23] performs inter-server scheduling by forwarding requests to the server

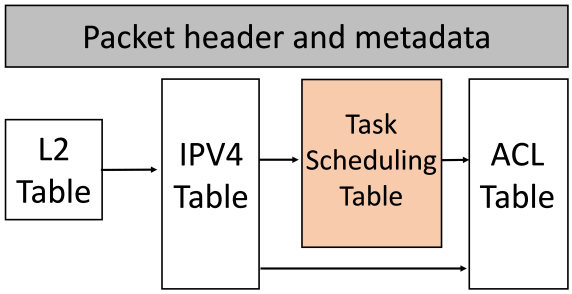
with the least load, the request is then scheduled on the appropriate worker on that server using another intra-server scheduler. Racksched supports request affinity by mapping request IDs to the scheduled server in a `ReqTable`, when any remaining packet from a request is sent Racksched checks the `ReqTable` to forward them to the same selected server. Due to the nature of their scheduling algorithm Racksched does not scale the number of servers well. Racksched needs to select the server with the minimum load and can only compute a limited number of load minimums per stage, needing more stages the more servers are available. To avoid head-of-lines blocking Racksched samples `k` servers before computing the minimum queue using a tree-based mechanism and scheduling the request to the server with the shortest queue.

2.3 Programmable Switches

Programmable switches, such as Barefoot's Tofino [19, 20] and Broadcom's Trident 3 [30], are network-programmable ASICs which help with implementing custom packet processing pipeline that can run at line speed.



(a) Switch data plane.



(b) Pipeline for implementing in-networking scheduling

Figure 3: Switch data plane

Figure 3a illustrates the basic data plane architecture of modern programmable switches. A packet first goes through the ingress pipeline, then the traffic manager handles the packet and forwards it to the egress pipeline, it is then processed by the egress pipeline before it is finally emitted.

Each pipeline is composed of multiple stages (Figure 3b). with each stage, a packet or metadata fields match with one or more tables. The appropriate action is then executed. Each stage has its own resources, this includes tables and memory registers. Data can be shared between stages using packet headers and small per-packet metadata (a few hundred bytes in size) that is propagated with the packet as it goes through the pipeline (Figure 3b). Packet processing can be viewed as a graph of match-action stages.

Programmers use domain-specific programming languages such as P4 [31, 32] to implement their custom packet processing protocols, actions and tables, and define their custom packet headers.

Challenges. Modern ASICs are restricted due to their need to process packets at line speed. They are limited by (1) Number of stages per pipeline, (2) number of memory accesses per stage, (3) number of tables and register per stage, (4) the size of data that can be read/written per packet per stage, and (5) the size of meta data per packet. In addition, modern ASICs lack support for loops or recursion.

The ASICs limiting memory model is particularly challenging to overcome when implementing an in-network scheduler. A register is the only way to store data that can be accessed between packets, but it can only be accessed once per stage using a single operation. This operation can be either a read, a write, a simple arithmetic operation (e.g., read and increment or read and set), or a simple logical operation.

3 Falcon Overview

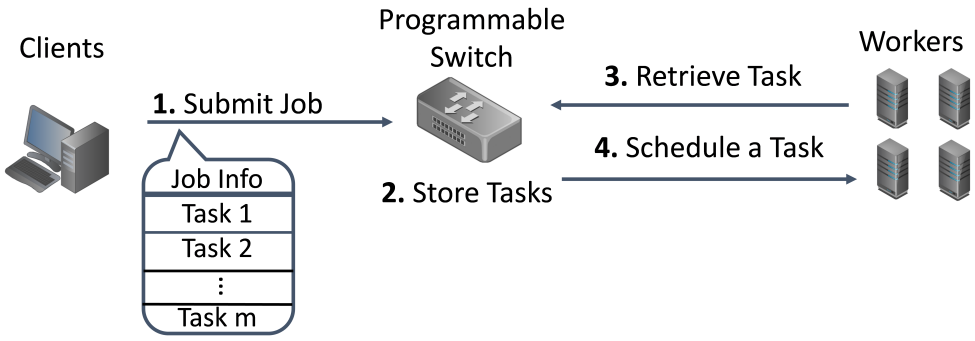


Figure 4: Falcon's Architecture

Falcon is a centralized scheduler running on a programmable switch. Falcon places tasks on free executors in the cluster with low overheads. We identify a process executing work assigned to it by the scheduler as an executor. Multiple executor processes are launched on a single node. The number of executors running on a node is usually limited by the number of logical cores available on that node. Figure 4 shows the basic architecture of Falcon consisting of clients, executors, and a centralized programmable switch.

3.1 Falcon Client

Falcon client batches a number of independent tasks and forwards it to the scheduler as a job for submission, this is similar to how Spark [12] and Sparrow [17] group and submit their tasks. The client can also be referred to as the data-processing framework. In this

paper, we use the terms client and *data-processing framework* interchangeably. Following the current structure of data processing frameworks, a group of independent tasks able to run in parallel is known as a job. Clients manage these tasks' data dependencies between different jobs [11, 12, 17]. Clients are responsible for tracking and resubmitting any failed or lost tasks [12, 17].

3.2 Executors

Figure 2 shows Falcon's scheduling timeline. Executors request new tasks from the scheduler whenever an executor is free. This way the scheduler only schedules tasks on free executors eliminating any head-of-line blocking. Making it easier to achieve our low-latency requirement. The executor reports back to the scheduler when it completes its assigned task, which is then forwarded to the client informing it of the completion.

If no tasks are available on the scheduler, the scheduler sends an invalid task back to the executor. The executor would then resubmit its request after a configurable amount of time.

3.3 Programmable Switch

A programmable switch [19, 20] hosts our Falcon in-network implementation. The scheduler stores tasks it receives from our clients (Figure 4) in a circular queue before an executor is free and requests a task to run. This task queue stores all the information needed to link the tasks to their client. The scheduler chooses and forwards tasks to the executors when they are free and request them. The tasks are chosen based on different requirements depending on the policy the scheduler is running (i.e., FIFO, or locality).

Implementing such simple scheduling design on a programmable switch is difficult due

to the various limitations of modern programmable switches.

3.4 Deployment Approach

Similar to previous projects which make use of programmable switches and their capabilities [33, 34, 35, 36] our network controller reroutes all job-submission packets through the single programmable switch that runs our Falcon scheduler. These packets are rerouted by installing forwarding rules to direct them to the switch. It's possible that this approach will forward packets through a longer path than traditional forwarding, but the increase in latency is minimal. Li et al. [34] report that for 88% of cases, using this method results in no additional request latency, while the 99th percentile show added latency less than 5 µs.

3.5 Fault Tolerance

The scheduler maintains a soft state. We address switch failures by replacing it with another switch to run the scheduling algorithm. The clients track any tasks lost during this failure and resubmit them to the new switch. This is similar to other frameworks [11, 12, 17] where the client times out for failed tasks and reissues a job submission with these tasks.

This means the client or executor will also resend lost job-submission packets or task-completion packets. This may result in tasks being executed multiple times. This does not affect correctness due to tasks being idempotent [11], but has a small effect on cluster utilization efficiency and latencies.

4 Falcon Design

We first present the base design for Falcon's scheduler with a FIFO scheduling policy, then extend this design with data locality-ware (§5) scheduling.

4.1 Network Protocol

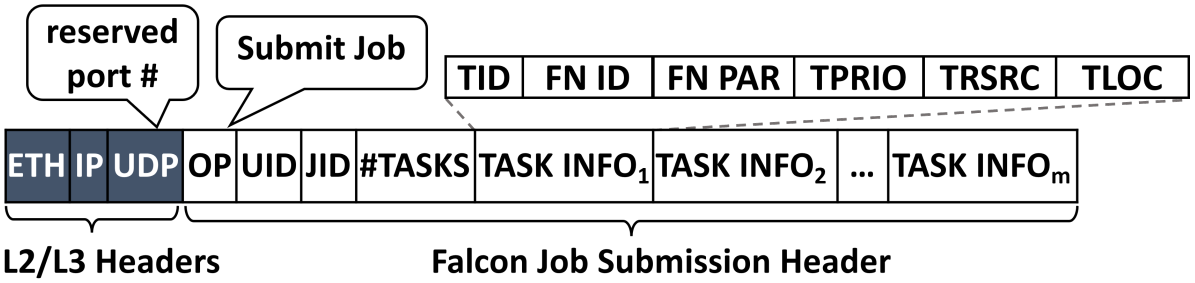


Figure 5: Falcon's job submission packet.

Falcon uses UDP to employ an application-layer protocol embedded in the L4 payload. Other systems that use programmable switches [33, 34, 35] also use UDP to lower latencies.

Falcon presents two new packet headers: `job_submission` packet, which is used to send task batches as jobs to the scheduler, and a `retrieve_task` packet that is used to request tasks from the scheduler and send tasks to executors. We detail these headers next and use the next subsections to discuss our design.

Figure 5 shows the `job_submission` packet structure:

- `OP`: This field indicates that this packet is a job submission request.

- `UID`: A unique user identifier.
- `JID`: A per client unique job ID. The <`UID`, `JID`>are used as unique identifier for each job.
- `#TASKS`: The number of tasks in the job. This is used by the scheduler to parse through this packet and store all the tasks.
- `TASK_INFO`: metadata for each task in the job submission packet.

The task information (`TASK_INFO`) contains:

- `TID`: A per job unique task identifier. The tuple <`UID`, `JID`, `TID`>is a used as a unique identifier for each task.
- `TDESC`: This decides what type of task to run, and any parameters needed for its execution.
- `TLOC`: data locality information. `TLOC` lists the nodes id and rack id of nodes holding task's data.

The scheduler uses a `retrieve_task` packet to place a task and send it to an executor. The `retrieve_task` packet includes client information and `TASK_INFO` of the task being retrieved.

4.2 Scheduler Design

Falcon uses the switch registers as a circular queue to store tasks. The following is stored in each queue entry: `TASK_INFO`, `client_IP`, and `client_port`, and an `is_valid` field to

indicate if this task was already scheduled or not. The circular queue has two 32-bit pointers: `add_ptr` and `retrieve_ptr`. The `retrieve_ptr` points to the first valid task in the queue and the `add_ptr` points to next empty entry in the queue where a new task is to be stored.

Each pointer comprises two parts: <`round_num`, `index`>. The index indicates which entry in the queue this pointer points to. The `round_num` counts how many times the pointer went through the whole queue. This field is used to solve special cases when the queue is full or empty.

We subtract `retrieve_ptr` from `add_ptr` to calculate if the queue is full or empty. If the result is zero, then we have an empty queue. If the subtractions result is equal or larger than the queue size, then we have a full queue. If the difference is negative, then the `retrieve_ptr` is larger than `add_ptr` and we need to fix these pointers. We discuss this below.

When implementing a simple circular queue, to insert a new task, we check if the queue has space by subtracting the `retrieve_ptr` from `add_ptr`. If the queue has space, we insert the new task in the queue and increment the `add_ptr`. However, implementing such design on current programmable switches is not possible. This is because `add_ptr` is accessed twice; once when determining if the queue is full and the second is when it increments. Dequeuing a task faces a similar problem.

Because a packet is limited to only one access for each memory unit, Falcon uses an atomic `read_and_increment()` operation on `add_ptr` to read and increment it in one access. Falcon then adds the new task into the queue if it is not full. Using this atomic operation increments the `add_ptr` even if the queue is full. Falcon uses the same atomic operation to increment `retrieve_ptr` when dequeuing a task even if the queue is empty. In these cases,

we need to fix the pointers. We achieve that by using another packet so we can access these pointers again. This is later explained in section 4.5.

4.3 Handling Job Submission

The client generates and populates a `job_submission` packet (Figure 5), then sends it to the scheduler for submission. The scheduler stores the tasks in the queue.

Inserting tasks into the circular queue can be challenging since programmable switches lack support for any type of loops or recursion and the queue can only be accessed once per packet. Falcon solves this problem by first checking how many tasks are in the packet by reading the `#TASKS` field. If there are any tasks, Falcon first extracts the first task from the list of `TASK_INFO`, and then enqueues the task into the queue while incrementing the `add_ptr`.

Enqueuing Multiple Tasks. To enqueue all the tasks in the `TASK_INFO` field (Figure 5), Falcon uses packet recirculation (i.e., emitting a packet from egress to ingress through a special port and processing it as if it is a new packet). The scheduler extracts the first task from the `job_submission` packet, decrements the `#TASKS` field, then emits the packet through the recirculation port. Falcon repeats this process until the `#TASKS` field reaches zero.

Handling a Full Queue. With each task in a `job_submission` packet, Falcon calls `read_and_increment(add_ptr)`. Falcon then checks the difference between `add_ptr` and `retrieve_ptr` to check if the queue is full and inserts the task if it is not full. If the queue is full, Falcon generates an error packet and forwards it to the client. This packet details which tasks were not submitted, meaning they are not in the queue. The client resubmits these tasks later using another `job_submission` packet.

4.4 Handling Task Retrieval

Whenever an executor is free it requests a task from the scheduler. This helps avoid head-of-line blocking. The executor requests a task by sending a `retrieve_task` packet to Falcon. The scheduler executes `read_and_increment(retrieve_ptr)` upon receiving a `retrieve_task` packet, then it dequeues and reads a task from the queue. The scheduler then checks the task's `is_valid` flag. If the flag is true, then the scheduler forwards the task to the executor to run and sets the `is_valid` flag to false (This can be done in only one access by using `read_and_set(is_valid, false)`). Alternatively, if the `is_valid` flag is false then the queue is empty. The scheduler then sends a no-op task back to the executor and the executor would wait a configurable amount of time before requesting a task again.

4.5 Pointer Correction

Falcon calls `read_and_increment(add_ptr)` on every `job_submission` packet before checking if the queue is full or not. If the queue is full, then the `add_ptr` value is incorrect and it needs to be fixed. In this case, the scheduler generates and recirculates a repair packet to fix the `add_ptr` value. Since Falcon is running on a pipelined model, we need to avoid generating multiple repair packets. Falcon uses a Boolean flag (`is_repairing_add_ptr`) to determine whether any repair packet is recirculating to reset the `add_ptr`, this way only one repair packet is generated.

Similarly, the scheduler calls `read_and_increment(retrieve_ptr)` on every retrieve task operation before checking if the task is valid or not. If the task is invalid (meaning the queue is empty), then the `retrieve_ptr` is incorrect and needs to be reset. We fix this pointer when we receive the next `job_submission` packet. Upon receiving the next `job_submission` packet the scheduler enqueues the first task into the queue. Falcon then

determines if the `retrieve_ptr` is incorrect and needs fixing (i.e., This is done by checking if the `retrieve_ptr` is ahead of the `add_ptr`). If the `retrieve_ptr` is incorrect, Falcon recirculates a packet to set the `retrieve_ptr` to point to the newly added task. Similar to the way we fix the `add_ptr`, we make use of the `is_repairing_retrieve_ptr` Boolean to recirculate only one repair packet.

5 Locality-Aware Scheduling

We adopt locality aware scheduling semantics of Spark [12]. Falcon support multiple levels of locality. Each task is tagged with node ids 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 nodes holding task's data are not free, the scheduler will try to place the task on a node in the same rack as the nodes holding the data. If that is not successful, the task is placed on any node in the cluster. Similar to current frameworks, clients tag tasks with data location information.

Similar to Spark, Falcon maintains a `skip_counter` that counts the number of times a task is considered for scheduling. This additional field is stored in the task queue (Section 4.1). Falcon has two configuration parameters: `node_limit` and `rack_limit`. If the `skip_counter` is larger than `node_limit` the task will be considered for scheduling on the nodes in the same rack as the nodes holding its data. If the `skip_counter` is larger than `rack_limit` the task will be scheduled on the next available worker regardless of data locality.

Job submission. Clients use the `TLOC` field in the `TASK-INFO` (Section 4.1) to specify nodes holding tasks' data. The job submission packets is processed as described in Section 4.3.

Task retrieval. When processing a task retrieval request, the scheduler retrieves a task from the queue as described in Section 4.4. The scheduler then checks the TLOC field of the task, if the task is local to the worker, the task is scheduled at the worker, else, the

task's skip_count is incremented.

If the retrieved task is not local to that worker and skip_count is smaller than node_limit, Falcon reinserts it back into the queue and picks up the next task in the queue. To do this we switch the retrieved task TASK_INFO with the next valid task TASK_INFO.

Swapping a Task. We recirculate our populated swap_task packet. The swap_task packet contains these fields: TASK_INFO which contains info of the retrieved task; SWAP_IDX, this contains the index of the next task in the queue; EXEC_ID, this contains the ID of the executor; the IP address and port of the executor; and the current retrieve_ptr value.

When a swap_task packet arrives at the switch, the scheduler swaps the task in the packet with the task in the queue at the index SWAP_IDX field from the swap packet. the scheduler does not increment the retrieve_ptr when processing swap_task packets.

If the swapped task has data local on the executor specified in EXEC_ID, the task is forwarded back to the executor. Otherwise, the scheduler increments the skip counter of the task and the SWAP_IDX field in the swap_task packet. the scheduler then recirculates this swap_task packet and recirculates this packet.

The scheduler repeats this swapping process until a task's TLOC field matches the EXEC_ID field or the swap_task packet traverses the whole queue. The scheduler checks if the SWAP_IDX is ahead of the add_ptr, this means that the swap packet traversed the whole queue and could not find a task to run on the executor. The scheduler then enqueues the task in the swap_task packet back into the queue before sending an invalid task to the executor. The executor, upon receiving an invalid task, would send a new retrieve_task packet after a short timeout period.

The swap_task packet contains the value of the retrieve_ptr before swapping. When the retrieve_ptr on the packet does not match the current retrieve_ptr, the sched-

uler swaps the task in the packet with the task pointed at by the current `retrieve_ptr` rather than the task at index `SWAP_IDX`. This is done because other retrieve packets have incremented the `retrieve_ptr` and now the `SWAP_IDX` points towards an invalid task.

Skip counter. Every task in the queue starts with a strict locality requirement where they are only scheduled on preferred nodes. But if a task is skipped enough times to where its `skip_counter` reachs a preconfigured `node_limit` then the locality requirement is loosened. Tasks that reach the `node_locality_limit` can now be scheduled on nodes that occupy the same rack as the preferred node. If a task is still not scheduled and its `skip_counter` reaches a `rack_limit` then this task can be assigned to any node in the cluster and is scheduled to the first worker that requests a task.Both the `node_limit` and the `rack_limit` are configurable.

6 Implementation

We implemented our Falcon workers, clients, and software server using C++. Our Falcon scheduling algorithms were implemented using P4 [31, 32] to run on our Barefoot Tofino switch [20]. Our P4 implementation has 1500 lines of code. We adopt the same job submission model as Sparrow [17]. Our C++ client submits jobs every fixed interval. This interval and other task information for the submitted jobs can be changed to target different types of workloads.

We use an early Tofino switch model. The Tofino programmable switch has limited capabilities. Newer models of the Barefoot Tofino switch has more memory and stages [19]. Our Falcon prototype uses a task queue that can fit 128k tasks, This is due to the switch's limited capabilities. We divide our 32 bit queue pointers into a round_num which is represented by the 15 MSB, and an index for the 17 LSB. After doing some rough calculations, We suspect that the new generation of switches can have queues the size of about one million tasks.

6.1 DPDK

We implemented our workers, clients, and software server on top of DPDK [37] instead of cpp raw sockets. First we tried using ANS[38] a DPDK Native Accelerated Network Stack before moving to using a pure DPDK solution.

DPDK ANS. First we tried using ANS for our implementation. ANS does not support blocking for its sockets, so we moved to an epoll based solution. This ANS based solution

achieved around 325k task throughput per second with 60 threads while our previous cpp sockets implementation capped out at 700k task throughput per second with 60 threads. To try and fix our DPDK ANS solution we :

- Increased the number of available cores for ANS and our ANS based code.
- Increased the size and number of memory channels.
- Increased the number of hugepages and buffer sizes.
- Isolated cores that are running ANS and our ANS based applications.
- Increased the TXs and RXs batch sizes.
- Pad thread counters.

None of the above solutions worked, so we moved our implementation directly to DPDK after ANS failed to perform better than cpp sockets with our cluster configuration. We can not inspect ANS as it is not open source.

Pure DPDK. We have implemented our workers, clients, and software server using DPDK in 2500 lines of code. Every thread is run on a separate logical core and each have their own TX and RX queues. We bind each of our RX queues to a specific port then we direct incoming packets to the appropriate port using rte_flow api. We also make use of the rte_ring api in our software server for a lock-less ring implementation.

6.2 Simulation

We started from the Sparrow simulator code base [17] and built an event- based simulator for Falcon. We used our simulator to evaluate Falcon's performance at scale since we do

not have access to a cluster with Tofino switch that can stress our P4 Falcon scheduler implementation.

Th simulator processes events from a priority queue where events are sorted based on their start time. After an event is processed the simulator generates the next event and calculates its start time by adding the appropriate delays to the previous event's end time. We added Executor and Scheduler limits to the simulator to more accurately simulate high throughput scenarios.

7 Evaluation

We perform our experiments and compare Falcon's performance against the performance of state-of-the-art schedulers.

Testbed. We use our cluster of 13 nodes to run our experiments. Each node has 48GB of RAM, an Intel Xeon Silver 10-core CPU, and a 100 Gbps Mellanox MT27800 Connectx-5 NIC. We use an Edgecore Wedge switch with a Barefoot Tofino ASIC [20] to connect these nodes. Depending on the experiment, the switch runs either a simple forwarding program or a scheduler's P4 implementation. 10 of our cluster nodes are used as worker nodes, while the remaining are used for running schedulers or clients.

Alternatives. We run our experiments on the following schedulers.

- **Sparrow.** Sparrow [17] Samples a number of executors to find a free executors to place tasks one. We evaluated Sparrow's implementation [39] and found that it is not implemented efficiently because of their use of Java and RPCs. We re-implemented Sparrow using C++. Our Sparrow C++ implementation supports up to 25 times more throughput and achieves 2 times lower scheduling overhead than the original Sparrow.For our experiments, we use our Sparrow C++ implementation.

- **Falcon.** We use our Falcon P4 implementation.

- **Falcon-Server.** We use our Falcon C++ implementation which we run on one of our nodes. This implementation is a highly optimized centralized scheduler running our Falcon scheduling algorithm.

- **R2P2.** We use R2P2 P4 implementation. We use an executor queue size of 3 as suggested by the original paper [22].

Other Schedulers. We run our experiments on Spark [12] and found that Spark could not handle tasks that have sub-second service times; this is similar to Sparrow's [17] findings. Spark achieves a scheduling delay of 3 seconds with 50% cluster load. When going over the 50% load, Spark experiences infinite queuing and is not able to keep up with the workload. Considering our workloads mainly consist of micro-second tasks, We do not include Spark's delays in our figures. We run similar experiments on Firmament [13] using their open-source implementation. This implementation could not schedule and run tasks with service time lower than 1 second. Nevertheless, Gog et al. [13] note that Firmament is unable to support more than 1200 executors (running on 100 nodes) when using tasks with a service time of 5 ms. This is a scheduling throughput of roughly 240k request per second.

Workload. We use a synthetic benchmark as well traces from a Google cluster to compare Falcon to the alternatives. In all our experiments, we report the average of 10 runs. The standard deviation in all our experiments was under 5%.

7.1 Scheduling Latency

We compare the performance on Falcon to the state-of-the-art alternatives with a suit of synthetic benchmarks. We generate tasks with fixed execution time of 100 µs , 250 µs, 500 µs, bimodal (50% 100 µs and 50% 500 µs), trimodal (33.3% 100 µs, 33.3% 250 µs, and 33.3% 500 µs). We also experimented with execution times that follow an exponential distribution with a mean that follows the execution times presented above. We present the result with the exponential distribution with a mean execution time of 250 µs. The

evaluation with the exponential distribution of task execution time achieved comparable result to fixed execution times.

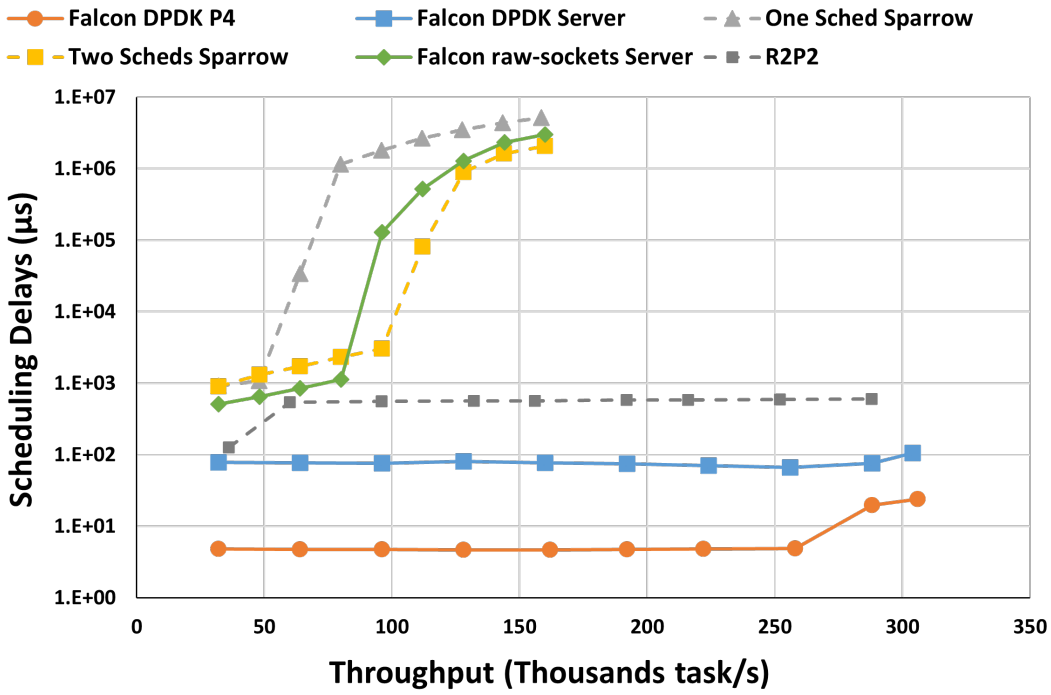
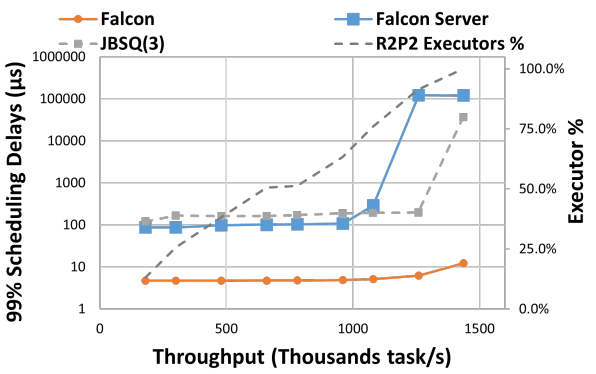


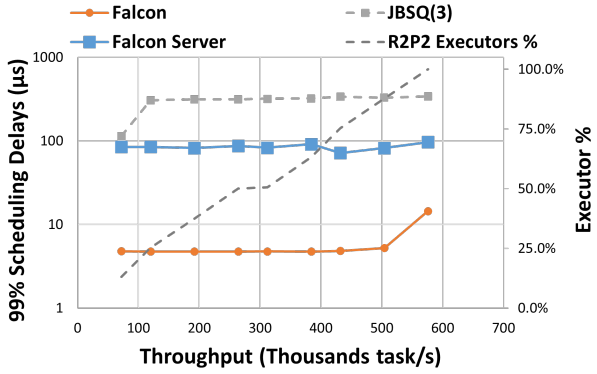
Figure 6: 99th Percentile scheduling delays with 500 µs tasks. Note the log scale on y axis.

Figure 6 shows the throughput and 99th percentile of the scheduling delay of six alternatives. We compare Falcon with DPDK (Falcon DPDK P4 in 6) to five alternatives: a C++ implementation running on a single server of Falcon using DPDK (Falcon DPDK Server) and POSIX sockets (Falcon Raw socket Server), R2P2 using DPDK (R2P2 DPDK P4), and Sparrow deployed with a single scheudler (One Sched Sparrow) and two schedulers (Two Sched Sparrow) built using POSIX sockets. The experiments use a synthetic workload with 500 µs CPU-intensive tasks. Every executor performs floating point integer calculations for the service time of the task.

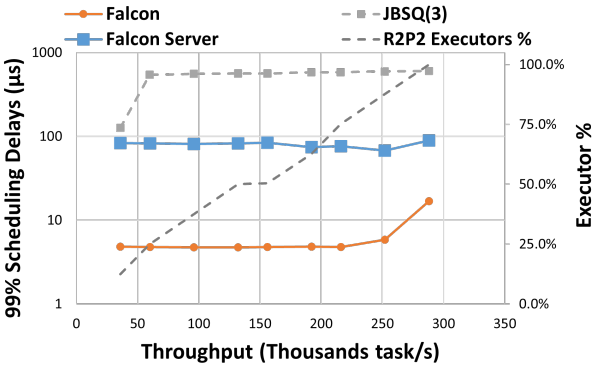
Figure 6 shows that Falcon significantly lowers the scheduling overhead compared to all other systems with a 20 µs tail latency, 120 times lower latency than that of the next alternative. We note that the tail latency increases with over 250 thousands tasks per second (tps). This is because the cluster utilization is over 90% and the tasks experience queuing delays. Nevertheless, even at high cluster utilization Falcon achieves 30 times lower latency than the closest alternative. We note that systems that use POSIX sockets can not support more than 160 thousand tps and achieve 100,000 times higher tail latency compared to Falcon. We note that Falcon C++ implementation achieves 1.7 times lower latency than a single Sparrow scheduler and a comparable performance to two Sparrow scheduler. Sparrow experiences higher latencies because of their reservation policy adding additional overhead, and because its implementation can not handle tasks with service times lower than 1 ms. Sparrow also might not schedule tasks to the ideal node as it probes part of the cluster for each task. These alternatives could not run workloads with lower execution times, consequently we do not present their results in the rest of the figures.



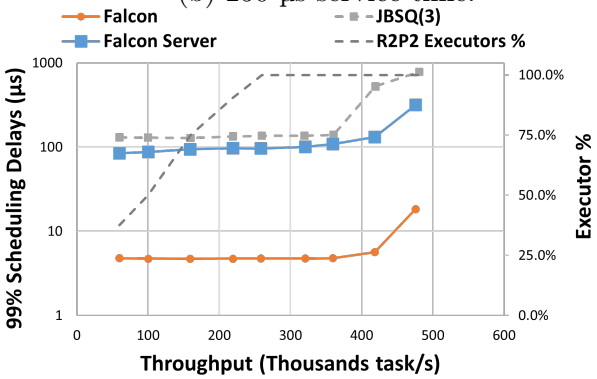
(a) 100 µs service time.



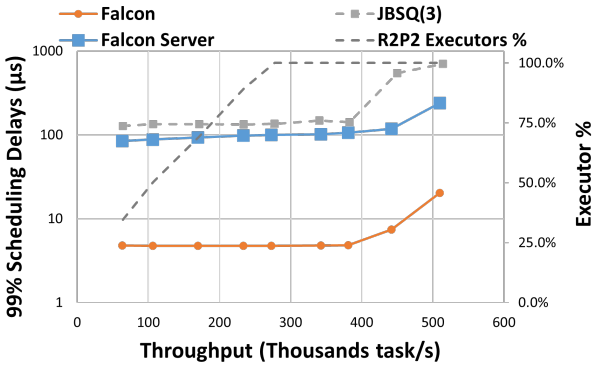
(b) 250 µs service time.



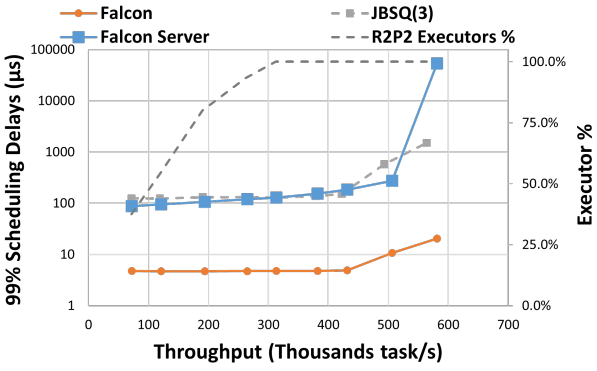
(c) 500 µs service time.



(d) Bimodal workload.



(e) Trimodal workload.



(f) Exp. workload.

Figure 7: 99$^{\text{th}}$ percentile of the scheduling delay. Note the log scale y axis.

Figure 7 shows the 99th of the scheduling delay with the suit of synthetic benchmarks. Falcon consistently achieves 4.7 - 20 µs scheduling delay, and achieves up to 1.4 Mtps scheduling throughput. We note that with 1.4 Mtps the cluster utization was 90%. R2P2 has 100 to 190 µs scheduling delay and can support up to 1.1 Mtps. At high loads R2P2 scheduling delay can reach 36 ms (Figure 7.a). R2P2 experiences higher latencies than Falcon due to its heavy reliance on recirculation while Falcon only recirculates on job submissions.

7.2 Scheduling Throughput

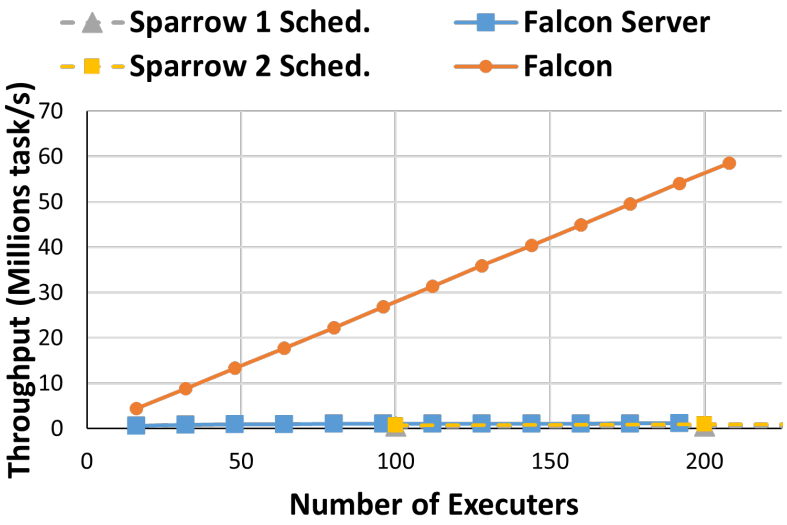


Figure 8: Scheduling throughput

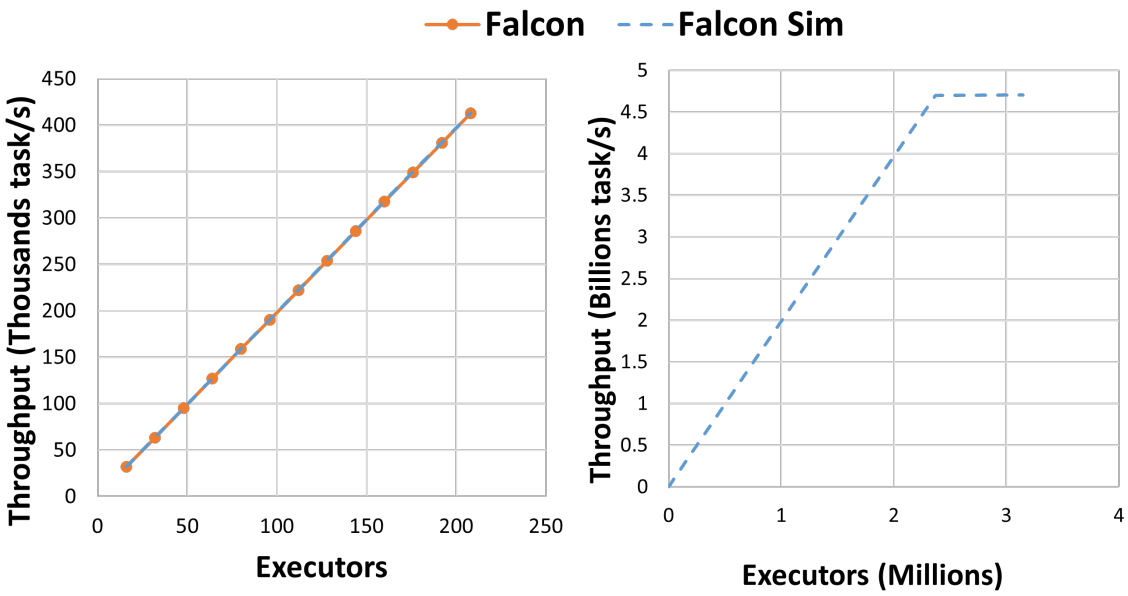
To measure the throughput of the scheduler, we generated a synthetic workload composed of zero service time tasks. This means executors request and receive a task, instantly finish it, and then request another one. We stress the schedulers by increasing the number of executors, which increases the number of requests. For measuring Sparrow's throughput,

we skip the probing and sampling step from their protocol. This favours Sparrow as it eliminates the probing step's overhead. This means Sparrow's results are closer to its upper limit in terms of performance.

Figure 8 shows the scheduling throughput when scaling up the number of executors. Falcon 's performance improves linearly with additional executors. Falcon achieves 58 Mtps throughput with 208 executors. Unfortunately, we could not stress Falcon by deploying more executors on our cluster. The switch can support up to 4.7 billion packets per second according to its data sheet. This means to measure Falcon's throughput limit, we need to generate a workload substantially greater than what our executors can generate. We evaluate Falcon 's performance at a larger scale through simulation in §7.3. We note that all other systems do not scale well. Among the server -based schedulers, Falcon-Server has the highest throughput of 1.1 Mtps. Sparrow throughput is the lowest at 500 Ktps for a single Sparrow scheduler and 900 Ktps for two schedulers.

7.3 Falcon at Scale

Unfortunately, we could not stress the Falcon scheduler with our cluster. To evaluate Falcon 's performance at scale, we resort to simulation. We started from the Sparrow simulator code base [39] and built an event-based simulator for Falcon. Our simulator receives the peak scheduler throughput as input and uses it to simulate scenarios with varying executor counts and task durations. For Sparrow, we set the scheduling throughput to 500 K decisions per second per scheduler. This is the throughput measured by profiling the Sparrow scheduler on our cluster. For Falcon, our switch data sheet states that the switch can process 4.7 billion packets per second [20]. We use this limit in our simulator.



(a) Simulator validation. The lines for Falcon and Falcon simulation overlap. The difference between the two is less than 5%.

(b) Falcon's throughput at scale with 500 µs tasks.

Figure 9: Simulator validation and Falcon's throughput simulation.

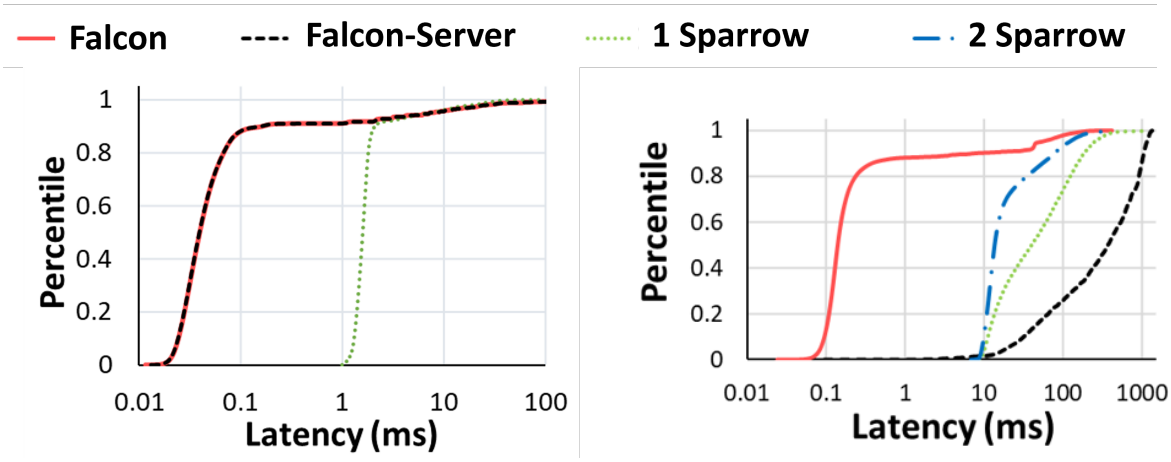
Simulator Validation. To validate the simulator, we compare the scheduling throughput of our simulator and the real execution of the cluster when running jobs with 500 µs tasks. Figure 9a compares the scheduling throughput of the real execution to the simulation run for Falcon. The figure shows that the simulator is highly accurate in computing the scheduling throughput of Falcon. In the worst case, the simulation results are only 5% different from the real workload at high loads. We also validated the simulator results with no-op tasks and observed similar results.

Simulation of Large Clusters. We used the simulator to evaluate Falcon 's ability to support large clusters when running 500 µs tasks. Figure 9b shows the throughput of

Falcon on clusters with millions of executors. The figure shows that at its peak scheduling throughput, Falcon can support clusters with 2.3 million executors. Note that these executors usually represent logical cores. If we assume every physical core supports two logical cores, our results indicate that Falcon can scale to a cluster with over a million cores.

7.4 Performance with Real Workload

We evaluate the scheduler performance using Google cluster traces [40]. The Google traces include information for tasks running on a 12,500-node cluster at Google over a month. To run this trace on our 12-node cluster, we use an approach similar to Firmament's [13]. We accelerate a uniform sample from the Google trace to generate a trace that can finish executing on our cluster in 3 minutes. We vary the sampling rate according to the number of executors. The generated trace for this experiment has a median task service time of 5 ms.



(a) CPU-intesive workload

(b) Sleep workload

Figure 10: Scheduling latency CDF. X-axes are in log-scale.

Figure 10a shows the CDF of total scheduling delays for multiple scheduling alternatives running CPU-intensive Google trace tasks. Executors perform floating point integer operations for the service time of these tasks. Using this workload, we can run 16 executors per worker on our 11 worker nodes for a total of 176 executors. Looking at Figure 10a we see that Falcon achieves 40 times lower scheduling delays than the next alternative. Falcon achieves a median scheduling delay of 0.04 ms, while a single Sparrow scheduler achieves 1.6 ms median scheduling delay. We explain later in section 7.5 the reason for difference in performance. Both of our Falcon implementations show comparable results. This is because our server implementation is lightweight and highly optimized. Falcon and Falcon-Server experience scheduling delays that are dominated by network delays as a result of the get_task() operation. Falcon and Falcon-Server achieve latencies of over 5 ms at the 95th percentile due to bursty nature of the workload, where hundreds of tasks that share the same submission time experience long queueing delays. We detail this further using a breakdown of the scheduling overhead in the next section.

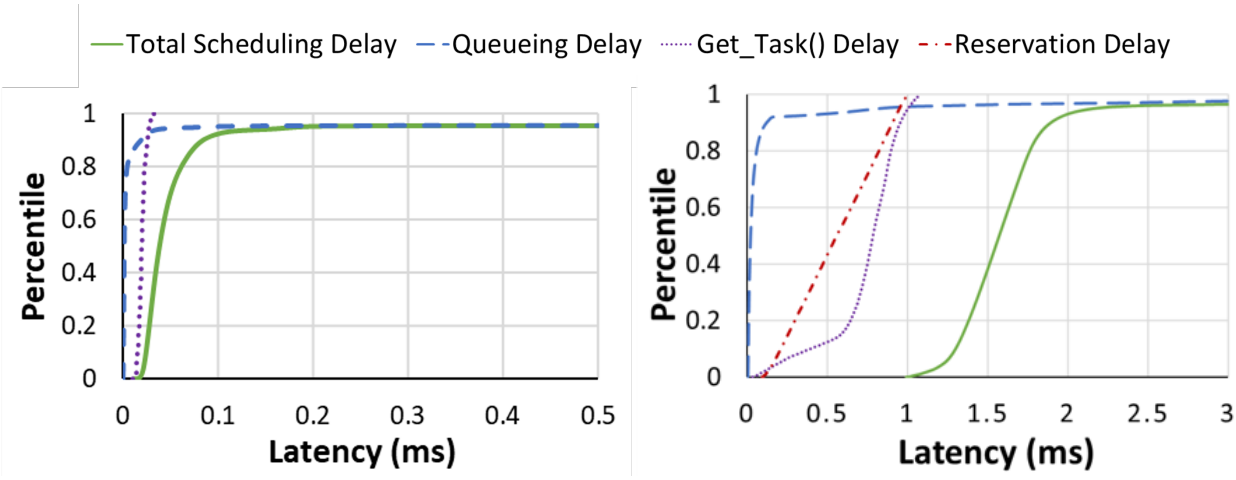
To achieve higher throughput, we switch our executors to sleep rather than running floating point operations for the service time of the tasks. This way we can launch more executors. We launch 200 executors per worker on our 11 worker nodes for a total of 2200 executors. Figure 10b shows the CDF of total scheduling delays for different schedulers running the sleep-based Google trace tasks. Falcon achieves at least 100 times lower scheduling delays than the next alternative. Falcon reaches a median delay of 0.14 ms, while other scheduling alternatives achieve over 14 ms of median scheduling delays. Our Falcon-server implementation performs poorly compared to other alternatives; a single Sparrow scheduler achieves a median of 40 ms while Falcon-server's median is 370 ms. This is because Sparrow executors request tasks only when they are probed, while Falcon-Server executors keep polling the scheduler for tasks. Falcon executors re-send their get task request after

waiting 50 µs when receiving a no-op task. This 50 µs wait time works for Falcon but strains our Falcon-Server implementation.

7.5 Scheduling Overhead Breakdown

We time each step of the scheduling protocol to better understand the reason for the difference in performance between the different systems. The protocols steps are shown in Figures 1 and 2. Figure 11 plots the CDF breakdown of scheduling overhead for the experiment in Figure 10a. This experiment uses floating point integer operations Google trace tasks on a 176-executor cluster.

Figure 10a shows Falcon with 40 times lower median scheduling delays than the next alternative and 13 times lower at the 90^{th} percentile. Both Falcon and Sparrow achieve similar latency at the 95^{th} percentile. This is because of the bursty nature of the trace, where hundreds of tasks share the same submission time. These tasks are queued for a longer time waiting for the next free executor.



(a) Falcon breakdown

(b) Sparrow breakdown

Figure 11: Scheduling overhead breakdown. Note that the x-axis is in a different scale in (a) and (b).

Looking at Figure 11a we see that Falcon's queueing delay dominates its total scheduling delay. 90% of Falcon 's get_task() delays are equal to the network round trip time. Figure 11b shows that 37% of Sparrow's scheduling delay consists of reservation delay (The delay for the probing step in their protocol), while the rest is credited to get_task() delays and queueing delays. Looking at Sparrow's implementation we found that it employs different threads for receiving tasks and for scheduling them. The threads communicate with each other using a shared queue that is accessed using a global lock. Sparrow's executors also employ different threads for processing and running tasks. These threads also communicate with each other using a shared queue that is accessed using a global lock. This implementation of Sparrow and the resulting thread contention adds to the get_task() and reservation delays.

7.6 Locality-Aware Scheduling

To test our locality-based scheduling we desinged an experiments that emulates a mutli-rack deployment. We divided our 10 worker nodes into 3 racks. Each node runs 16 executors.We set the round trip time for intra rack communication to 20 µs and inter rack RTT to 100 µs [41]. We run a CPU-intensive synthetic locality aware workload with 100 µs tasks. The processed data is not replicated and is evenly partitioned across the nodes. Consequently, each task has it data local on one node in the cluster.

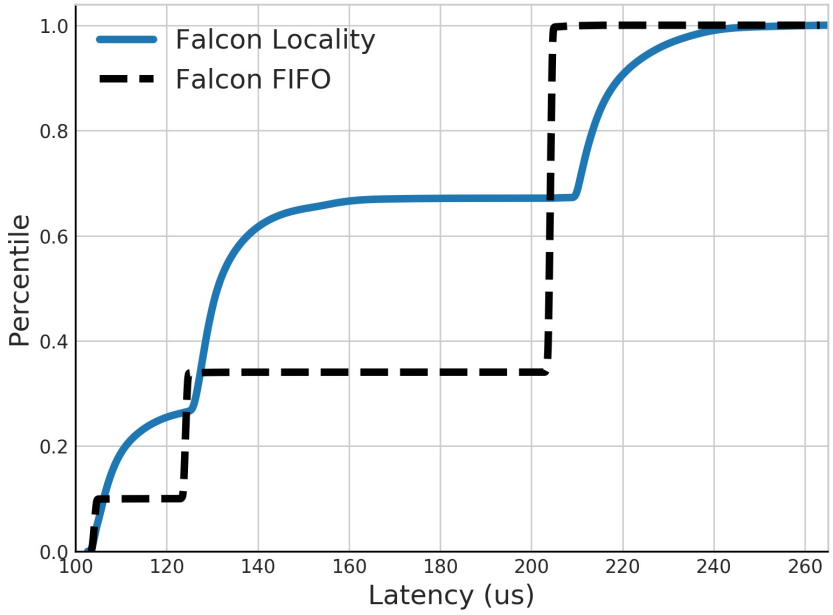


Figure 12: End to end latency CDF for 3:9 locality limits vs FIFO.

We run this workload with different configurations of `node_locality_limit` and `rack_locality_limit`. Figure 12 shows the CDF of the scheduling delay with a `node_locality_limit` of 3 and a `rack_locality_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 were on another rack. Our FIFO design placed 10.03% of tasks on their preferred node, 24.05% on the same rack, and the last 65.94% on a different rack. We experiments with other values for these two limits and noticed that in all configurations at least 49% of tasks are scheduled on the target node or rack.

Figure 12 shows the total end to end delay CDF of tasks scheduled using our locality aware scheduler vs our FIFO scheduler. We note that FIFO has median latency of 203.87 µs while Falcon locality has a median latency of 31.35 µs. Falcon locality performs 2x better at the 66^{th} percentile, after that FIFO achieves lower latencies due to incurring the same locality overhead as Falcon locality without the need to recirculate packets.

8 Related Work

Hybrid Scheduling. Hawk [16] and Mercury [42] 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 real-time tasks.

Streaming Systems. Numerous systems [43, 44] have been designed to tackle sub-second tasks in the streaming environment. However, they do not target tasks in a parallel compute environment and do not target tasks in the microsecond range.

Network-Accelerated Systems. Many recent projects have used programmable switches to accelerate consensus protocols [33, 34, 45, 46] and implement in-network caching [47], DNN training and inferencing [48], and in-network aggregation operations [49]. JumpGate [50] proposed offloading some data analytics functions to the switch. It did not investigate supporting in-network scheduling. R2P2 [22] proposes a scheduling approach that leverages programmable switches. However, it only supports JBSQ scheduling and does not support a locality aware policy.

Low-Latency. Several projects have explored operating system and network stack optimizations for low latency workloads. These efforts include using kernel-bypass techniques [51, 52, 37] and efficient core reallocation mechanisms [53]. These efforts are orthogonal to ours as we explore a scheduler architecture that can support real-time analytics on large clusters.

9 Concluding Remarks

We present Falcon, a centralized in-network scheduler that can assign tasks to the next available executor at line-rate and scale to process billions of requests per second. Our evaluation shows that Falcon can reduce scheduling overheads by an order of magnitude and achieve significantly higher throughput when compared to current state-of-the-art low-latency schedulers. Furthermore, our evaluation shows that a single switch can save tens of nodes in the cluster that would run schedulers under the distributed scheduling approach. Falcon demonstrates that despite their strict programming and memory model, modern programmable switches can be leveraged to implement complex scheduling policies.

References

[1] 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.*, 2020.

[2] 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.

[3] 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.

[4] Tan Zhang, Aakanksha Chowdhery, Paramvir Bahl, Kyle Jamieson, and Suman Banerjee. The design and implementation of a wireless video surveillance system. *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 426–438, 2015.

[5] 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.

[6] Stephen F. Elston and Melinda J. Wilson. Big data and smart trading. Retrieved from https://dsimg.ubm-us.net/envelope/86703/367082/1348004765_SYB_Big_Data_and_Smart_Trading_WP_Apr2012_WEB.pdf.

[7] 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.

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

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

[10] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 2019.

[11] 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.

[12] 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.

[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] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Tofino-2 second-generation of world's fastest p4-programmable ethernet switch asics. Retrieved March 16, 2021 from https://www.barefootnetworks.com/products/brief-tofino-2/.

[20] Tofino world's fastest p4-programmable ethernet switch asics. Retrieved March 16, 2021 from https://www.barefootnetworks.com/products/brief-tofino/.

[21] Cisco. Cisco nexus 34180yc and 3464c programmable switches data sheet. Retrieved January 16, 2021 from https://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/datasheet-c78-740836.html.

[22] 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.

[23] 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.

[24] A. I. Maarala, M. Rautiainen, M. Salmi, S. Pirttikangas, and J. Riekki. Low latency analytics for streaming traffic data with apache spark. *IEEE International Conference on Big Data (Big Data)*, page 2855–2858, 2015.

[25] 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.

[26] Y. Yan, Y. Qian, H. Sharif, and D. Tipper. A survey on smart grid communication infrastructures: Motivations, requirements and challenges. *IEEE Communications Surveys Tutorials*, 15(1):5–20, 2013.

[27] M. Tortonesi, A. Morelli, M. Govoni, et al. Leveraging internet of things within the military network environment — challenges and solutions. *IEEE 3rd World Forum on Internet of Things (WF-IoT)*, 3:111–116, 2016.

[28] O. Elijah, T. A. Rahman, I. Orikumhi, C. Y. Leow, and M. N. Hindia. An overview of internet of things (iot) and data analytics in agriculture: Benefits and challenges. *IEEE Internet of Things Journal*, 5(5):3758–3773, 2018.

[29] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous real-time object recognition on mobile devices. *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 155–168, 2015.

[30] Trident3-x7 / bcm56870 series. Retrieved January 21, 2021 from https://www.broadcom.com/products/ethernet-connectivity/switch-fabric/bcm56870.

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

[32] P4. Retrieved January 24, 2021 from https://p4.org/.

[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. pages 31–44, 2016.

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

[38] Ans-dpdk native accelerated network stack. Retrieved April 20, 2022 from https://www.ansyun.com/.

[39] Sparrow git repository. 2013. Retrieved January 23, 2021 from https://github.com/radlab/sparrow.

[40] John Wilkes. Google clusterdata 2011 traces. GitHub. Retrieved January 16, 2021 from https://github.com/google/cluster-data.

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

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, 2009.

[55] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM 2008 Conference on Data Communication, Association for Computing Machinery*, page 63–74, 2008.

[56] John Ousterhout, Arjun Gopalan, Ashish Gupta, et al. The ramcloud storage system. *ACM Transactions on Computer Systems*, 33(3):1–77:55, 2015.

[57] Rajesh Nishtala, Hans Fugal, Steven Grimm, et al. Scaling memcache at facebook. *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), USENIX Association*, pages 385–398, 2013.

[58] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. Enabling flexible network fpga clusters in a heterogeneous cloud data center. *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Association for Computing Machinery*, pages 237–246, 2017.

[59] A. Putnam, A. M. Caulfield, E. S. Chung, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, 2014.

[60] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast analytical power management for latency-critical systems. *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 598–610, 2015.

[61] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. *ACM SIGARCH Computer Architecture News*, 42(3):301–312, 2014.

[62] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4):42–47, 2005.

[63] S. Verma, Y. Kawamoto, Z. M. Fadlullah, H. Nishiyama, and N. Kato. A survey on network methodologies for real-time analytics of massive iot data and open research issues. *IEEE Communications Surveys & Tutorials*, 19(3):1457–1477, 2017.