Network-Accelerated Scheduling for Large Clusters

by

Ibrahim Sami Kettaneh

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, 2020

Author’s Declaration

Statement of Contribution

The materials of this thesis have been either published or submitted for publication. The author of this thesis was the leader of all the projects presented in this thesis performing most or all of the design, implementation, and evaluation. He also led the writing effort and co-authored the corresponding papers. Below are the details for each chapter, this section presents the corresponding publications and details the role of the author of this thesis.

- Chapter 3: The findings of this chapter are under review at HotCloud 2020. The author of this thesis was the main contributor for this project and its publication starting from the initial idea, system development, and evaluation. Sreeharsha Udayashankar and Ashraf Abel-hadi helped with surveying the related work and evaluating Spark.

  Related paper:

  - **Ibrahim Kettaneh**, Sreeharsha Udayashankar, Ashraf Abdel-hadi, Samer Al-Kiswany "Falcon: Low-Latency, Network-Accelerated Scheduling" in 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20) (Under review)

- Chapter 4: The findings of this chapter are published at NSDI 2020. The author of this thesis did the switch pipeline implementation, the controller implementation, and conducted most of the experiments in the evaluation section. Hatem Takruri and Ahmed Alquraan helped with implementing the server-side (section 4.2 and 4.3).

  Related paper:

  - **Ibrahim Kettaneh**, Ahmed Alquraan, Hatem Takruri, Samer Al-Kiswany "Accelerating Reads with In-Network Consistency-Aware Load Balancing" in Volume 37, issue 1 of ACM Transactions on Computer Systems (TOCS) (Under Submission)
  - Hatem Takruri, **Ibrahim Kettaneh**, Ahmed Alquraan, Samer Al-Kiswany "FLAIR: Accelerating Reads with Consistency-Aware Network Routing," in 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), 2020, pp. 723-737. (the first two authors contributed equally to the project)

Summary of other projects conducted during my Master's study

In addition to these publications, I integrated the load balancing technique that is discussed in Chapter 4 with an object storage system called NICE. This effort led to one additional publication:

- **Ibrahim Kettaneh**, Ahmed Alquraan, Hatem Takruri, Suli Yang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Samer Al-Kiswany "The Network-Integrated Storage System" in IEEE Transactions on Parallel and Distributed Systems (TPDS) 2019.

Abstract

We explore a novel design approach for accelerating schedulers for large scale clusters. Our approach follows a centralized design and leverages the programmability of recent programmable switches to accelerating scheduling operations. We demonstrate the feasibility and benefits of this approach by building two schedulers: one for accelerating data analytics scheduling and one for accelerating scheduling in key-value stores.

First, we present a scheduler designed for low-latency data analytics workloads. The proposed scheduler receives job description, maintains a task queue in the switch memory, and schedules tasks on the next available worker at line-rate. The core of this design is a novel pipeline-based scheduling logic that can schedule tasks at line-rate. Our prototype evaluation on a cluster with a Barefoot Tofino switch shows that the proposed approach can reduce scheduling overhead by an order of magnitude compared to state-of-the-art schedulers.

Second, we present a network-accelerated scheduler for linearizable key-value stores. The proposed design exploits programmable switches to keep track of write requests and responses, and to identify where the latest version of each object is stored. Our prototype evaluation shows that the proposed design achieves up to 42% higher throughput, and 35-97% lower latency than the current state-of-the-art approaches.

Acknowledgements

I would like to start this by expressing my sincere gratitude to my advisor Prof. Samer Al-Kiswany for his support during my MMath study, his immense knowledge, patience and motivation. He gave me the help I needed all the time during my study. I could not have imagined a better mentor and advisor.

I would like to thank Prof. Khuzaima Daudjee, and Prof. Martin Karsten for their valuable feedback.

I would like to thank my lab mates for their support and for the fun we had in the last two years. I would like to specially thank Ahmed Alquraan, Mohammed Alfatafta, and Hatem Takruri for the stimulating discussions we had, and for the sleepless nights we had before deadlines. Also, I would like to thank my friends Feras Moussally and Abdelrahman Allaf for their continuous support despite the physical distance. I would like to than Ashraf Abdel-hadi, and Sreeharsha Udayashankar for their fruitful collaboration in this project.

I would like to thank my family: My parents, my brother and my sisters for supporting me throughout my life, I cannot thank you enough. I would like to thank my nieces and nephews for their continuous concerns about how I am doing here, you surprised me how quickly you grew up in just two years.

Last but not least, I would like to thank everyone who helped me in this journey, and I forgot to mention their names.

Dedication

I dedicate this to my parents for their endless love, encouragement, and support.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Modern computer clusters consist of thousands of nodes that can perform millions of operations per second [1, 2]. A key to utilizing these large clusters is to assign tasks accurately and efficiently to nodes. An ideal scheduler can accurately dispatch tasks to free workers, maintain a low scheduling delay, scale to thousands of nodes, and incorporate complex scheduling policies, such as data location-aware scheduling.

Realizing this ideal scheduler eluded the research community for decades, modern schedulers targeting large clusters adopted a centralized scheduling approach [3, 4]. In this design, a single scheduler monitors the load on all cluster nodes, maintains information about all the scheduled and ready-to-schedule tasks, and performs all scheduling decisions. As the central schedule has complete information about the cluster load, it is able to make accurate scheduling decisions. In particular, it can schedule the next task on the first node that becomes free. However, centralized schedulers have a major shortcoming; it cannot support large clusters because it cannot keep up with monitoring hundreds of cluster nodes and scheduling millions of tasks per second. Consequently, a centralized scheduler often presents a scalability bottleneck and imposes a high scheduling overhead.

To overcome the limitation of centralized schedulers, a number of projects explored a decentralized scheduling approach [2, 5], in which multiple schedulers work on assigning tasks to worker nodes. To scale this approach to a large number of schedulers, this approach avoids coordinating the decisions of the different schedulers and avoids monitoring the cluster load. To facilitate making scheduling decisions, each scheduler depends on approximate information about the cluster load, which is collected through sampling or through lazily collecting status reports from workers. As a result, schedulers use incomplete or stale data to make their scheduling decisions, which leads to inferior scheduling decisions.

In addition, a common scheduling optimization is to schedule a task on the node that stores the input data for that task. To enable the optimization, the scheduler needs to maintain accurate information about the location of the most recent version of every data object in the system. Unfortunately, maintaining this information in a timely manner is challenging in systems in which the data changes often (e.g., key-value storage systems).

In this thesis, we explore a new approach for designing a scheduler for large-scale clusters that can achieve the aforementioned goals. The proposed approach uses a centralized scheduling approach, in which a central scheduler monitors the system load and data location and uses this information to make accurate scheduling decisions. To overcome the limitations of a single node scheduler, we explore techniques to leverage programmable switches to accelerate the scheduler. Modern programmable switches present ideal candidates for accelerating scheduling for large-scale clusters. They can be programmed to run application-specific logic and can process billions of packets per second, a rate that is more than enough to support scheduling for a cluster with hundreds of nodes.

To explore this frontier, we explored the design of two schedulers: Falcon (Chapter 3) and FLAIR (Chapter 4).

Falcon is a network-accelerated scheduler for large-scale data analytics frameworks. In Falcon, we explore techniques to build a low-overhead scheduler for large clusters. Falcon uses a centralized design to identify which workers are free and, at line-rate, schedule tasks on free workers. The Falcon scheduler receives a task's specification and queues tasks in a switch register. To eliminate head-of-line blocking, workers poll the switch for new tasks. Despite its simplicity, implementing this approach is complicated due to the limitations of programmable switches (Chapter 2).

To demonstrate the powerful capabilities of the proposed approach, we built a Falcon prototype based on Sparrow, a state-of-the-art low latency scheduler, and built the scheduler using P4. The evaluation on a cluster with a Barefoot Tofino switch [6] shows Falcon can reduce scheduling overhead by up to 23 times compared to Sparrow, and for short tasks, it improves the task execution time by 25%.

FLAIR is a network-accelerated scheduler for linearizable key-value storage systems. In FLAIR, we explore building a scheduler to keep track of where the latest version of each data item resides and balance read requests across consistent replicas. The core of FLAIR is a packet-processing pipeline that maintains compact information about all objects stored in the system. FLAIR tracks every write request, and the corresponding system reply, to identify which objects are stable (i.e., not being modified) and which nodes in the cluster hold a consistent value for each object. Then, it uses this information to forward reads of stable objects to consistent followers.

To demonstrate the powerful capabilities of the proposed approach, we prototyped FlairKV, a key-value store built atop Raft [7]. Our evaluation of FlairKV shows FLAIR brings up to 2.8 times higher throughput than an optimized Raft implementation; at least 4 times higher throughput compared to

Viewstamped Replication, Raft, and FastPaxos; and up to 42% higher throughput and up to 35–97% lower latency for most workloads compared to a state-of-the-art leases-based design [8, 9].

The rest of the thesis is structured as follows. Chapter 2 discusses programmable switch capabilities and limitations. Chapter 3 presents Falcon, showing its design, implementation; and an evaluation of Falcon against the state-of-the-art low-overhead scheduler. Chapter 4 presents the design, implementation, and evaluation of FLAIR. Chapter 5 surveys the related work. We conclude in Chapter 6.

Chapter 2
Background

The proposed approach leverages the capabilities of modern programmable switch to accelerate scheduling for large scale cluster. In this section, we present an overview of the capabilities of modern programmable switches and discuss their limitations.

Programmable switches allow the implementation of an application-specific packet-processing pipeline that is deployed on network devices and executed at line-rate. A number of vendors produce network-programmable ASICs, including Barefoot's Tofino [10], Cavium's XPliant [11], and Broadcom Trident 3 [12].

Figure 1 illustrates the basic data plane architecture of modern programmable switches. The data plane contains three main components: ingress pipelines, a traffic manager, and egress pipelines. A packet is first processed by an ingress pipeline before it is forwarded by the traffic manager to the egress pipeline that will finally emit the packet.

A set of switch ports (say 16 ports) share one processing pipeline, and each pipeline is composed of multiple stages. At each stage, one or more tables match fields in the packet header or metadata; if a packet matches, the corresponding action is executed. Programmers can define custom headers and metadata as well as custom actions. Each stage has its own dedicated resources, including tables and register arrays (a memory buffer). Figure 2(a) shows a simple example of a pipeline that routes a request to a key-value store based on the key, and Figure 2(b) shows the details of the KV routing stage. The stage forwards the request based on the key in the packet's custom L4 header. The programmer

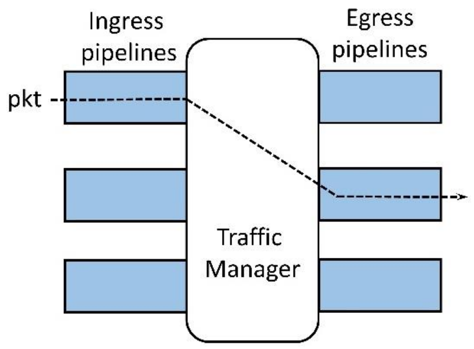
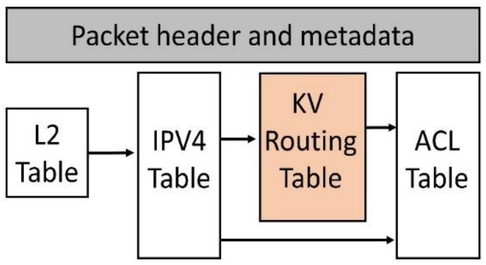
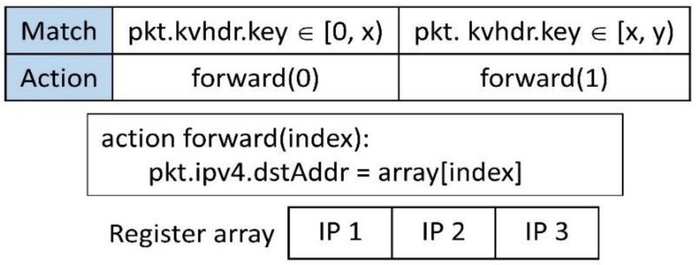


Figure 1. Switch pipelines



(a) Pipeline for routing based on a hash-based key.



(b) A simple match-action stage for routing based on a hash-based key for the KV routing table in subfigure (a)

Figure 2. Switch data plane.

implements a `forward()` action that accesses the register array holding nodes' IP addresses. An external controller can modify the register array and the table entries.

Stages can share data through the packet header and small per-packet metadata (a few hundred bytes in size) that is propagated between the stages as the packet is processed throughout the pipeline (Figure 2 (a)). The processing of packets can be viewed as a graph of match-action stages.

Programmers use domain-specific languages like P4 [13] to define their own packet headers, define tables, implement custom actions, and configure the processing graphs.

Challenges. While programmable ASICs and their domain-specific languages significantly increase the flexibility of network switches, the need to execute custom actions at line speed restricts what can be done. To process packets at line speed, P4 and modern programmable ASICs have to meet strict resource and timing requirements. Consequently, modern ASICs 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/written per-packet per register, (5) the size of per-packet metadata that is passed between stages. Finally, modern ASICs lack the support of loops or recursion.

The restrictive memory model constitutes a particular challenge to building an in-network scheduler. A given memory register (the only memory that can preserve variables across packets) can only be accessed in a single stage and using a single operation. The operation can be either a simple read or write or an atomic operation (e.g., read and increment or read and set).

Chapter 3
Falcon: Low Latency, Network-Accelerated Scheduling

Recent increased adoption of real-time analytics [14, 15] is pushing the limits of traditional data processing frameworks [16]. Applications such as real-time object recognition [17], real-time fraud detection [14], IoT applications [14], and video quality prediction [18] require processing millions of events per second and aim to provide a processing latency of a few milliseconds.

To support very short tasks that take tens of milliseconds, the scheduling throughput must be quite high. For a cluster of one thousand 32-core nodes, the scheduler must make more than 3 million scheduling decisions per second. Furthermore, for such tasks, scheduling delays beyond 1 ms are intolerable.

Traditional data processing frameworks use a centralized scheduler [3, 4]. Although the centralized scheduler has the most accurate knowledge about the utilization of each node in the cluster and can make accurate scheduling decisions; that is, they will schedule the head of the queue task on the first worker that becomes available. Unfortunately, this approach cannot scale to process thousands of status reports from cluster nodes and millions of scheduling decisions [18, 19].

To overcome the limitations of a centralized scheduler, low-latency data analytics engines [2, 5, 19, 20] have adopted a distributed scheduling approach. This approach employs tens of schedulers to increase scheduling throughput and reduce scheduling latency. As these schedulers do not have accurate information about the load in the cluster, they probe a randomly-selected subset of nodes to find nodes to run a given set of tasks [2, 5, 19]. The disadvantage of this approach is that the scheduling decisions are suboptimal, as they are based on partial information, and the additional probing step increases the scheduling delay. For instance, Sparrow [2], the state-of-the-art distributed scheduler, reports that, after excluding the queueing delay, the 50th percentile of the scheduling delay is 2 ms, and the 90th percentile is 10 ms. Our evaluation (Section 3.5) confirms Sparrow's high scheduling overhead. While this overhead is negligible when scheduling 100-ms tasks [2, 16], it is intolerable for short, 10-ms, tasks [21].

We therefore present Falcon, a scheduling approach that can support large-scale clusters while significantly reducing scheduling latency. Unlike current low-latency schedulers, Falcon adopts a centralized scheduling approach to eliminate the probing overhead and make precise scheduling decisions. To overcome the processing limitations of a single-node scheduler, Falcon offloads the scheduler to a network switch. Recent programmable switches (e.g., Barefoot Tofino [6]) facilitate the implementation of application-specific logic in network switches and can forward over 5 billion packets per second [6], making them ideal candidates for implementing a centralized scheduler.

Falcon leverages the power and flexibility of the new generation of programmable switches [6, 22] to design an in-network scheduler. The Falcon scheduler receives a task's specification and queues the task in a switch register. To eliminate head-of-line blocking, workers poll the switch for new tasks. This design leads to minimal scheduling overhead, as tasks are scheduled at a line-rate on the next free worker in the cluster. If no free worker exists, the task is queued at the switch.

Despite its simplicity, implementing this approach is complicated by the limitations of programmable switches. Modern programmable switches have a restrictive pipeline-based programmable model with limited computing capability and a restrictive memory model (Chapter 2). In particular, the restrictive memory model allows for performing a single operation on a memory location once per packet. Consequently, even implementing a simple task queue is complicated, as standard queue operations will access the queue size twice: once to check whether the queue is empty or full, and once to increment or decrement its size. Section 3.3 presents Falcon's novel task queue design.

To demonstrate the powerful capabilities of the proposed approach, we built a Falcon prototype based on Sparrow and built the scheduler using P4. Our evaluation on a cluster with a Barefoot Tofino switch [6] shows that Falcon can reduce scheduling overhead by up to 23× compared to Sparrow and for short tasks it improves the task execution time by 25%.

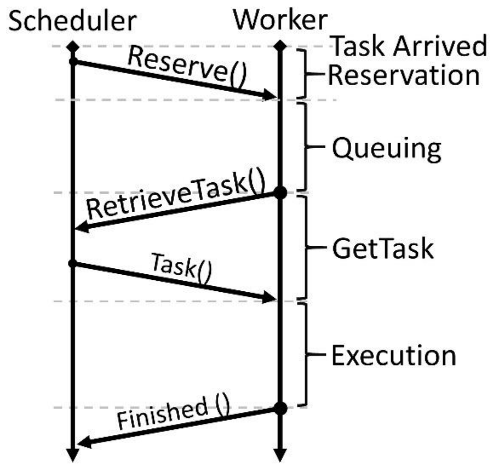
3.1 Background

In this section, we present an overview of the modern low-latency scheduler and discuss Sparrow's design.

3.1.1 Overview of Low-Latency Scheduling

Modern low-latency data analytics engines [2, 3, 5] follow a distributed scheduling approach. Large clusters include thousands of worker nodes that execute tasks and tens of scheduling nodes. A job consists of *m* independent tasks (*m* is typically a small number between 8 and 64), and any of the schedulers can handle a job.

In order to keep task execution cost low, real-time data analytics typically follow an executor model [2, 5]. Tasks are scheduled following a FIFO order. All workers have preloaded executables that represent different tasks. All workers have access to a shared storage system that hosts the input data.



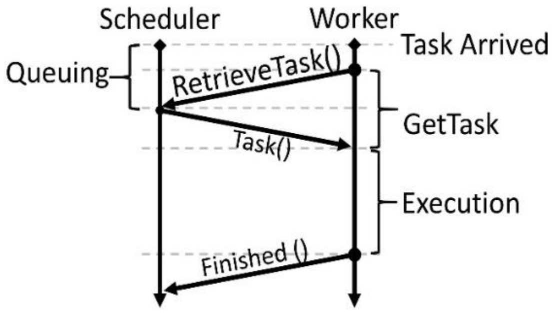


Figure 3. Sparrow's scheduling timeline. **Figure 4. Falcon's scheduling timeline.**

To schedule a task on a worker the scheduler sends the task id that unique identifies the operation, and an input id, which identifies the input data.

Figure 3 shows the scheduling steps in Sparrow. To schedule a job with *m* tasks, the scheduler submits probes to *2m* randomly selected workers. For instance, if the job has 32 tasks, the scheduler probes 64 out of, potentially, hundreds of nodes in the cluster. The workers queue the probes. When a worker completes its current task, it dequeues a probe, retrieves the task from the scheduler, and executes it. This probing technique is necessary, as the scheduler does not have complete knowledge of the cluster utilization. After completing *m* tasks, the scheduler proactively cancels the extra probes or discards future requests for task retrieval for those probes.

This approach has two shortcomings: first, as the scheduler only probes a small subset of nodes in the cluster, its scheduling decisions are inferior. Second, the probing step increases the scheduling latency.

3.2 Falcon Overview

Falcon is an in-network centralized scheduler that can assign tasks precisely to free workers with minimal overhead Figure 5 shows Falcon's architecture, which consists of worker nodes, client nodes, and a centralized programmable switch.

Client nodes. Similar to Spark [3] and Sparrow [2], clients group independent tasks into jobs and submit these jobs to the Falcon switch. Once all tasks in a particular job have finished executing, clients submit the next job. As in current data analytics frameworks, clients are responsible for tracking dependency between tasks and resubmitting failed tasks [2, 3].

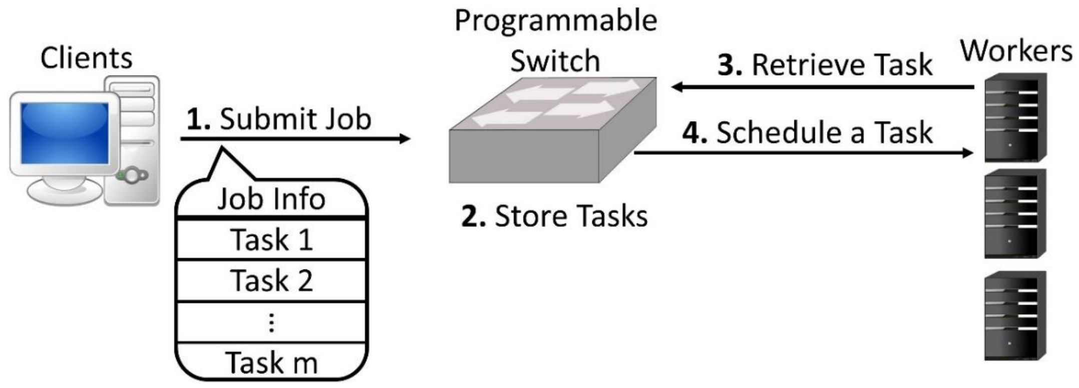


Figure 5. Falcon's architecture.

Worker nodes. Figure 4 shows the scheduling steps in Falcon. When a worker becomes free, it sends a message to the scheduler to request a new task. Thus, the scheduler only assigns tasks to free workers, effectively avoiding head-of-line blocking. If the scheduler has no tasks, it sends a no-op task to the worker, which requests a task after a short waiting period.

Programmable switch. Falcon uses a centralized in-network scheduler. The switch receives job descriptions that include a list of tasks (Figure 5). The switch adds these tasks to a circular FIFO queue. The switch assigns a task in first-come-first-serve order to the next worker that requests a task.

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

3.3 Falcon Design

3.3.1 Network Protocol

Falcon introduces an application-layer protocol embedded in packets' L4 payload. Similar to other systems that use programmable switches [23, 24, 25], Falcon uses UDP to reduce operation latency and simplify the scheduler design.

Falcon introduces two new packet headers: job_submission, which is used to submit a new job to the scheduler, and task_assignment packet used to send a task to a worker. We briefly discuss these headers in this section. The next subsections detail our design.

Figure 6 shows the main fields of the job_submission packet:

- OP*:* the request type: job submission or task assignment.
- UID: the user ID.
- JID: the job ID. The <UID, JID> combination represents a unique job identifier.

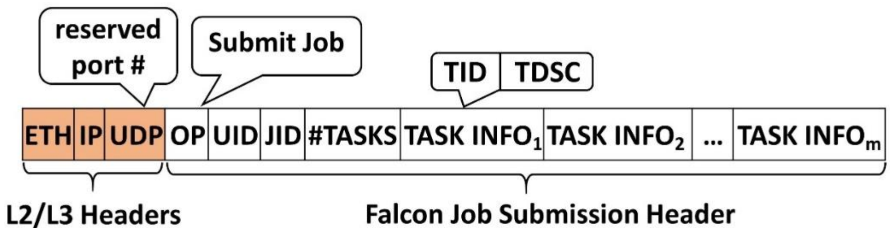


Figure 6. Falcon's job_submission header.

- #TASKS: the number of tasks in the job. The switch uses this field to parse the job submission packet properly.
- A list of TASK_INFO metadata for all the tasks in the job.

The task information (TASK_INFO) includes the following:

- TID*:* a task identifier within a job. The tuple <UID, JID, TID> is a unique identifier for all tasks submitted in the system.
- TDESC*:* the task description that determines the task to be executed.

To assign a task to a worker, the switch sends a task_assignment packet to the worker. The task_assignment header contains the TASK_INFO of a task, as well as the client IP address and port number.

3.3.2 Scheduler Design

Falcon stores tasks (i.e., TASK_INFO) in a switch register as a circular queue. Each queue entry has the following fields: TASK_INFO, client_IP, and client_port, as well as an is_valid flag that indicates whether the entry has been scheduled. The size of the queue in our implementation is 128K. The circular queue has two 32-bit pointers: add_ptr and retrieve_ptr. The add_ptr points to the next empty queue entry in which a new task can be inserted. The retrieve_ptr points to the next task to be scheduled in FIFO order.

Each pointer comprises two parts: <round_num, index>. The 17-bit index points to an entry in the queue. The 15-bit round_num counts the number of rounds the pointer traversed the entire queue. This round number helps to resolve special cases when the queue is full or empty.

To detect whether the queue is full or empty we subtract the retrieve_ptr from the add_ptr. If the difference is zero, the queue is empty. If the difference is equal to or larger than the queue size, the queue is full.

In some cases, the difference is negative, meaning the retrieve_ptr is larger than the add_ptr, in which case the pointers need an adjustment. We discuss this below.

In the standard circular queue implementation, 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, the new task is added to the queue and add_ptr is incremented. Unfortunately, this design cannot be implemented on current switches because it accesses add_ptr twice; it checks the pointer, then possibly increments it. The dequeue operation faces a similar challenge.

Because it can access a pointer only once per packet, Falcon uses an atomic read_and_increment(add_ptr) to read add_ptr and increment it in one access. It then checks whether the queue is full. If the queue is not full, Falcon uses the add_ptr value to retrieve a task from the queue. This approach increments add_ptr even when the queue is full. Similarly, to dequeue a task, Falcon calls read_and_increment(retrieve_ptr) and increments retrieve_ptr even when the queue is empty. In these cases, the pointers must be corrected, but because the pointer is accessible only once per packet, the correction must be made in some future packet. We discuss how to detect and correct queue pointers later in this section.

3.3.3 Handling Job Submission

The client submits a job by populating the header of a job_submission packet (Figure 6) and sending the packet to the switch. The switch then enqueues the job's 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, Falcon checks the #TASKS field in the packet. If it is larger than zero, it parses the first task in the packet's list of tasks, calls read_and_increment(add_ptr), then adds the task to the queue.

Adding Multiple Tasks. The job_submission packet (Figure 6) contains a list of tasks. To add multiple tasks to the queue, Falcon leverages packet recirculation, i.e., the ability to resubmit a packet from the egress pipeline to the ingress pipeline and process it as a new packet (Figure 1). The scheduler removes the first task from the task list in the job_submission packet, decrements the #TASKS field, and recirculates the packet. Falcon continues to recirculate the packet until #TASKS is zero.

Handling a Full Queue. When enqueueing a new task, the scheduler calls read_and_increment(add_ptr), then compares add_ptr and retrieve_ptr to determine whether the

```
1:   on_job_submission(task) {
2:       add_ptr_val = read_and_increment(add_ptr)
3:       retrieve_ptr_val = read(retrieve_ptr)
4:       if(queue has space) {
5:           push(task, add_ptr_val)
6:       }
7:       else if(queue is full){
8:           // add_ptr needs fixing
9:           a_fixing_val = read_and_set(is_repairing_add_ptr, true)
10:          if(a_fixing_val == false){
11:              // there no other attempt to fix the pointer.
12:              // set the add_ptr to its old value before the
13:              // increment.
14:              recirculate_fixing_packet(add_ptr, add_ptr_val)
15:          }
16:      }
17:      if(retrieve_ptr_val > add_ptr_val ){
18:           // the retrieve_ptr needs fixing
19:           r_fixing_val = read_and_set(is_repairing_retrieve_ptr, true)
20:           if(r_fixing_val == false){
21:              // there is no other attempt to fix the pointer.
22:              // set the retrieve_ptr to equal the add_ptr
23:              recirculate_fixing_packet(retrieve_ptr, add_ptr_val)
24:           }
25:       }
26:   }
```

Listing 1. Adding tasks pseudocode

queue is full. If the queue is not full, the scheduler adds the task to the queue. If the queue is full, the scheduler does not add the task and sends an error message to the client. The error message contains the list of tasks not added to the queue. The client then retries submitting a new job after a while.

Listing 1. Shows the pseudocode for job submission. We first extract a task from the packet (line 1). Then, we atomically read and increment the add_ptr (Line 2), and read the retrieve_ptr (line 3). Next, we compare these two pointers to determine the queue status. If the queue has space (line 4), we push the task to the add_ptr location (line 5). If the queue is full (line 7), and since we have incremented the add_ptr (line 2), we first set a boolean indicating that we detected that the add_ptr needs fixing (line 9), and we recirculate the packet to fix the pointer and set it where it was before this operation if the boolean was not already set (line 14). Similarly, if the retrieve_ptr needs fixing (line 17), we set a Boolean indicating that (line 19), and we recirculate a fixing packet to set it where to the add_ptr location if it was not already set (line 23).

```
1:    on_retrieve{
2:        rtrv_ptr_val = read_and_increment(retrieve_ptr)
3:        if(tasks[rtrv_ptr_val] is valid){
4:            pop(tasks[rtrv_ptr_val])
5:        }
6:        else{
7:           // oops! The queue is empty
8:           // retrieve_ptr needs fixing, it is done in
9:            // on_job_submission()
10:       }
11:   }
```

Listing 2. Retrieving tasks pseudocode

```

 on_fixing_ptr{
     ptr = header.ptr
     value = header.value
     if(ptr == retrieve_ptr){
         retrieve_ptr =  value
         is_repairing_retrieve_ptr = false
      }
     if(ptr == add_ptr){
         add_ptr = value
         is_repairing_add_ptr = false
      }
 }
```

Listing 3. Fixing pointers pseudocode

3.3.4 Handling Task Retrieval

To avoid head-of-line blocking, workers retrieve tasks only when they become free. A worker sends a request to the switch to retrieve a task. The scheduler calls read_and_increment(retrieve_ptr) and reads one task from the queue. If the task's is_valid flag is true, the task is sent to the worker, and the is_valid flag is set to false (this is done in one access with read_and_set(is_valid, false)). Otherwise, if the is_valid flag is false, this indicates that the queue is empty. In this case, the retrieve request is ignored, and the worker repeats the request after a while.

Listing 2. Shows the pseudocode for retrieving tasks. We first atomically read and increment the retrieve_ptr (line 2), if it is pointing to a valid task (line 3), we will pop it (line 4) and send it to the worker. If retrieve_ptr is pointing to an invalid task (line 6), it means that the queue is empty, and the retrieve_ptr needs to be fixed. The retrieve_ptr is fixed in the on_job_submission() function when the next job submission packet is received.

3.3.5 Correcting the Pointers

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 the add_ptr was a mistake. To correct this mistake, the scheduler recirculates a repair packet to reset the add_ptr to its original value. To avoid a case in which multiple job_submission packets try to reset the add_ptr, we added a Boolean flag (is_repairing_add_ptr) to ensure the scheduler only recirculates one repair packet.

Similarly, task retrieves operations call read_and_increment(retrieve_ptr), then check whether the retrieved task is valid. If the retrieved task is invalid (which indicates that the queue is empty), incrementing the pointer was a mistake. We leave this pointer until the next job_submission packet is received. On a job submission, if the retrieve_ptr is larger than add_ptr (which indicates that the retrieve_ptr needs repairing), the scheduler recirculates a packet and resets the retrieve_ptr to equal the index of the newly added task. We added a Boolean flag (is_repairing_retrieve_ptr) to ensure the scheduler only recirculates one repair packet.

Listing 3. Shows the pseudocode for fixing the pointers. Falcon uses recirculation to resubmit a packet to the switch pipeline (line 2). The resubmitted packet will determine the pointer that needs fixing and its value from the packet headers (lines 6-7). If the retrieve_ptr needs fixing, it will change its value (line 9), and set is_repairing_retrieve_ptr to false (line 10). If the add_ptr needs fixing, it will change its value (line 13), and set is_repairing_add_ptr to false (line 14).

3.4 Implementation

We implemented Falcon on top of Sparrow. We modified Sparrow to eliminate probing and to make workers send getTask() requests without having a prob. We implemented the Falcon pipeline and FIFO task queue using P4. Figure 7 Shows Falcon's pipeline implementation. First, if the switch detects a Falcon's packet, it checks the operation type (1). Depending on the operation, the switch will either change add_ptr or retrieve_ptr values (2-5). After that, the switch will check the status of the queue (6), and based on that, it performs pointers corrections if needed (7-9). Then, it will access the tasks buffer (either adding tasks or retrieving tasks) (10). Finally, if the packet is job submission, it will decrement the number of remaining tasks and will either recirculate it or drop it if there are no remaining tasks.

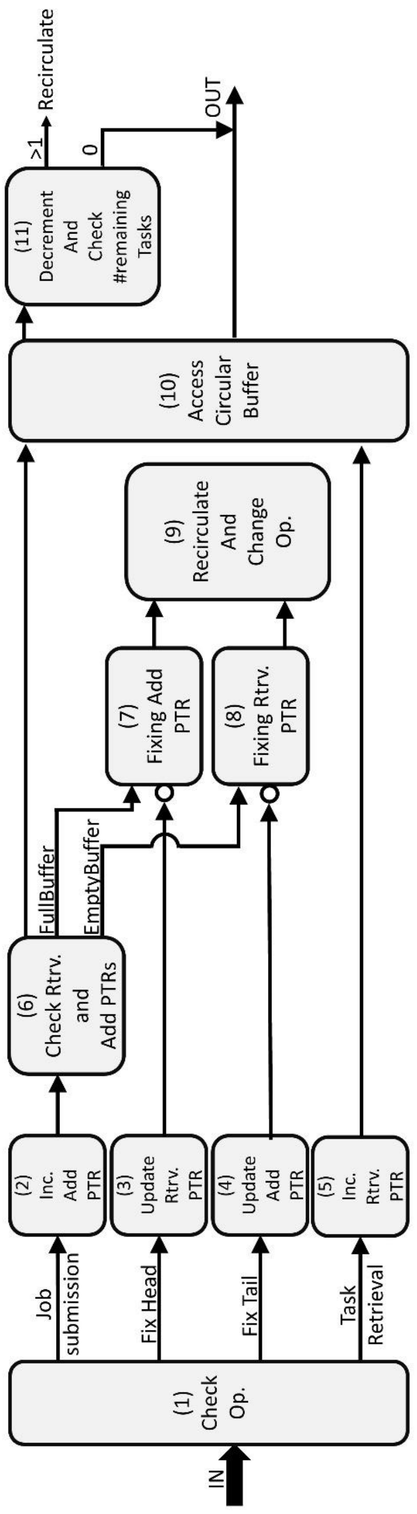


Figure 7. Falcon's pipeline

3.5 Evaluation

In this section, we compare the performance of Falcon against that of Sparrow, the current state-of-the-art distributed scheduler, and Spark, the state-of-the-art data analytics engine with centralized scheduling.

Testbed. We performed all experiments on a 12-node cluster. Each node had 48GB of RAM, an Intel Xeon Silver 10-core CPU, and a 100 Gbps Mellanox NIC. The nodes are connected by an Edgecore Wedge switch with a Barefoot Tofino ASIC. In all experiments, we used 10 nodes as worker nodes and 2 nodes as client nodes. Each worker node ran 6 executors (i.e., a total of 60 executors). For Sparrow, we ran two schedulers on the same client nodes, a configuration that is favorable to Sparrow because it reduces communication latency between the client and the collocated scheduler.

Workload. We used a synthetic workload similar to the one used to evaluate Sparrow [2]. Each one of the two clients submitted a job every 10 ms (for a total of 200 jobs per second), and each job contained a set tasks. Each tasks takes sleeps for 10 ms. We varied the number of tasks per job to change the system utilization.

Job Scheduling Delay. Figure 8 shows the job scheduling delays for various system utilization levels. Falcon significantly reduced the scheduling delay at all utilization levels. At the 95% utilization rate, Falcon reduced the median scheduling delay by 23× (0.09 ms compared to 2.22 ms for Sparrow) and the 95th percentile by 9.7× (0.25 ms compared to 2.68 ms for Sparrow). Even at 50% utilization, Falcon reduced the scheduling delay by up to 22×. Furthermore, unlike Sparrow, Falcon's median delays did not change as the utilization reaches 95% because Falcon can easily handle billions of packet and process requests at line-rate, whereas scheduling overhead increases in Sparrow with larger jobs and higher utilization.

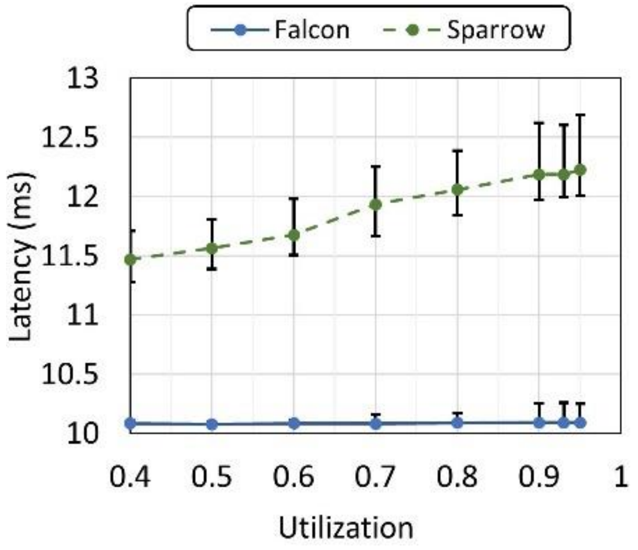


Figure 8. Job latency for various utilization rates. Error bars depict the 5th and 95th percentiles.

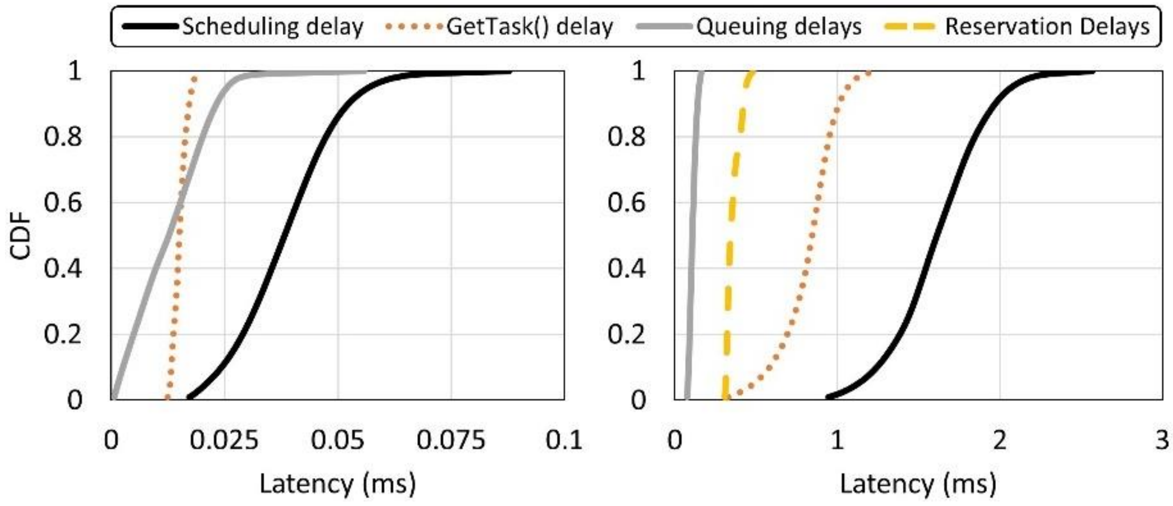


Figure 9. Breakdown of Falcon's scheduling delays. Note the difference in scale of the x-axis compared to Figure 10.

Figure 10. Breakdown of Sparrow's scheduling delays.

We also evaluated Spark's scheduling delay. Unfortunately, Spark did not scale well beyond 50% utilization: this confirms a similar observation made in the Sparrow paper [2]. The scheduling delay at 50% was 3 seconds. Above 50% utilization, the scheduler could not keep up and experienced infinite queueing. We did not include Spark in the figure for clarity.

Breaking Down the Scheduling Overhead. To understand the performance differences between Falcon and Sparrow, we measured the time spent on each step of the protocols (see Figure 3 and Figure 4). Figure 9 and Figure 10 show the CDF of every step of Falcon and Sparrow scheduling protocols, respectively, when running the system at 80% utilization. The figure shows the high impact of network acceleration. Falcon completed all scheduling steps in under 100 μs, whereas Sparrow took up to 2.5 ms. Although the reservation delay is unique to Sparrow, task retrieval and queueing delays are unavoidable regardless of the scheduling approach. Comparing the delay of these two steps shows that network acceleration brings up to 42× performance improvement. This significant performance improvement eliminates the need for multiple schedulers and shows that a single central scheduler can scale to support large clusters.

3.6 Summary and Future Work

We presented Falcon, a centralized in-network scheduler that can assign tasks to the next available worker at line-rate and scale to process billions of requests per second. Our evaluation shows that Falcon can reduce scheduling overhead by an order of magnitude compared to current state-of-the-art low-latency schedulers.

Compared to other projects that leverage programmable switches, Falcon explores the feasibility of offloading a complete service to the network. Falcon shows that, instead of running a scheduler on one or more servers, the complete scheduling service can be offloaded to the network.

In our current work, we are extending the scheduler to support data locality-aware scheduling and common scheduling constraints, such as scheduling on nodes with specific resources.

Chapter 4
FLAIR: Accelerating Reads with Consistency-Aware Network Routing

Replication is the main reliability technique for many modern cloud services [8, 26, 27] that process billions of requests each day [27, 28, 29]. Unfortunately, modern linearizable replication protocols [30] – such as multi-Paxos [31], Raft [7], Zab [32], and Viewstamped replication (VR) [33] – deliver poor read performance. This is because these protocols are leader-based: a single leader replica (or leader, for short) processes every read and write request, while follower replicas (followers for short) are used for reliability only.

Optimizing read performance is clearly important; for instance, the read-to-write ratio is 380:1 in Google's F1 advertising system [34], 500:1 in Facebook's TAO [29], and 30:1 in Facebook memcached deployments [35]. Previous efforts have attempted to accelerate reads by giving read leases [36] to some [37] or all followers [8, 9, 38], While holding a lease, a follower can serve read requests without consulting the leader; each lease has an expiration period. Unfortunately, this approach complicates the system's design, as it requires careful management of leases, affects the write operation – as all granted leases need to be revoked before an object can be modified – and imposes long delays when a follower holding a lease fails [8, 37].

We present the fast, linearizable, network-accelerated client reads (FLAIR), a novel protocol to serve reads from follower replicas with minimal changes to current leader-based replication protocols without using leases, all while preserving linearizability. In addition to improving read performance, FLAIR improves write performance by reducing the number of requests that must be handled by the leader and employing consistency-aware load-balancing.

FLAIR leverages the power and flexibility of the new generation of programmable switches. The core of FLAIR is a packet-processing pipeline (§4.4) that maintains compact information about all objects stored in the system. FLAIR tracks every write request and the corresponding system reply to identify which objects are stable (i.e., not being modified) and which followers hold a consistent value for each object, then uses this information to forward reads of stable objects to consistent followers. Followers optimistically serve reads and the FLAIR switch validates read replies to detect stale values. If the switch suspects that a reply from a follower is stale, it will drop the reply and resubmit the read request to the leader.

An additional benefit of FLAIR is that it facilitates the building of novel consistency-aware load balancing techniques. In systems that grant a lease to followers [8, 9, 37, 38], clients send read requests to a randomly selected follower. If the follower does not hold a lease, it blocks the request until it obtains a lease, or it forwards the request to the leader; either way, this approach adds additional delay. FLAIR does not incur this inefficiency as FLAIR load balances read requests only among followers that hold a consistent value for the requested object. we designed three consistency-aware load balancing techniques (§4.4.2): random, leader avoidance, and load awareness.

Despite its simplicity, implementing this approach is complicated by the limitations of programmable switches (Chapter 2) and the complexity of handling switch failures, network partitioning, and packet loss and reordering (§4.3.6).

To demonstrate the powerful capabilities of the proposed approach, we prototyped FlairKV (§4.4), a key-value store built atop Raft [7]. We made only minor changes to Raft's implementation [39] to enable followers to serve reads, make the leader order write requests following the sequence numbers assigned by the switch, and expose leader's log information to the FLAIR layer. The packet-processing pipeline was implemented using the P4 programming language [40]. We implemented the three aforementioned load-balancing techniques.

Our evaluation of FlairKV (§4.5) on a cluster with a Barefoot Tofino switch shows that FLAIR can bring sizable performance gains without increasing the complexity of the leader-based protocols or the write operation overhead. Our evaluation with different read-to-write ratios and workload skewness shows that FlairKV brings up to 2.8 times higher throughput than an optimized Raft implementation, at least 4 times higher throughput compared to Viewstamped replication, Raft, and FastPaxos, and up to 42% higher throughput and up to 35-97% lower latency for most workloads compared to state-of-the-art leases-based design [8, 9].

The performance and programmability of the new generation of switches open the door for the switches to be used beyond traditional network functionalities. We hope our experience will inform a new generation of distributed systems that co-design network protocols with systems operations.

4.1 Overview of Leader-based Replication Protocols

Leader-based consensus (LC) protocols [1, 7, 32, 33, 41, 42] are widely adopted in modern systems [9, 26, 27, 28]. The idea of having a leader that can commit an operation in a single round trip dates back to the early consensus protocols [31, 43]. Having a leader reduces contention and the number of messages, which greatly improves performance [31, 41].

LC protocols divide time into terms (a.k.a. views or epochs). Each term has a single leader; if the leader fails, a new term starts and a new leader is elected.

Clients send write requests to the leader (1 in Figure 11). The leader appends the request to its local log (2) and then sends the request to all follower replicas (3). A follower appends the request to its log (4) before sending an acknowledgment to the leader (5). If the leader receives an acknowledgment from a majority of its followers, the operation is considered *committed*. The leader *applies* the operation to its local state machine (e.g., in memory key-value store in Figure 11) in (6), then acknowledges the operation to the client (7). The leader will asynchronously inform the followers that it committed the operation. Followers maintain a *commit_index*, a log index pointing to the last committed operation in the log; when a follower receives the commit notification, it advances its *commit_index* and applies the write to its local store.

The replicated log has two properties that make it easy to reason about: it is guaranteed that if an operation at index *i* is committed, then every operation with an index smaller than *i* is committed as well; and if a follower accepts a new entry to its log, it is guaranteed that its log is identical to the leader's log up to that entry.

Client read requests are also sent to the leader. In Raft, the leader sends a heartbeat to all followers to make sure it is still the leader. If a majority of followers reply, the leader serves the read form its local store: it will check that all committed operations related to the requested object are applied to the local store before serving the request.

A common optimization is the leader lease optimization. Instead of collecting a majority of heartbeats for every read request, a majority of the followers can give the leader a lease [7, 41]. While holding a lease, the leader serves reads locally without contacting followers. Unfortunately, even with this optimization, the performance of the leader-based protocols is limited to a single-node performance.

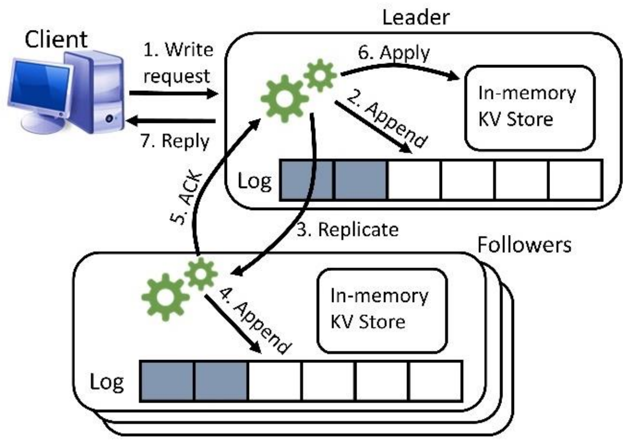


Figure 11. The path for a write operation.

4.2 FLAIR Overview

FLAIR is a novel protocol that targets deployments in a single data center. Figure 12 shows the system architecture, which consists of a programmable switch, a central controller, and storage nodes. Typically, multiple FLAIR instances are deployed with each serving a disjoint set of objects. For simplicity, we present a FLAIR deployment with one replica set (i.e., one leader and its followers).

FLAIR divides time into sessions (Figure 13). During a session the leader is bonded to a single switch that runs the FLAIR pipeline. Each session has a unique id that is assigned in a strictly increasing order. A session ends when a leader fails or the leader suspects that the switch has failed. An LC term may have one or more sessions, but a session does not span multiple terms.

A session starts with the FLAIR module at the leader (dubbed the *lflair* module) incrementing the session id, committing it to the LC log, updating the switch information about the objects in the system, then activating the session at the switch. *lflair* module keeps the switch's information up to date while in an active session. If the switch does not have an active session it drops all FLAIR packets.

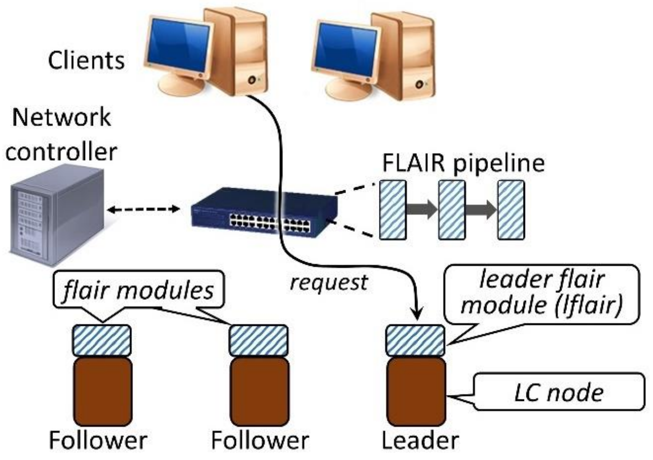


Figure 12. System architecture. The solid arrow shows a client request, while the dashed arrow show control messages.

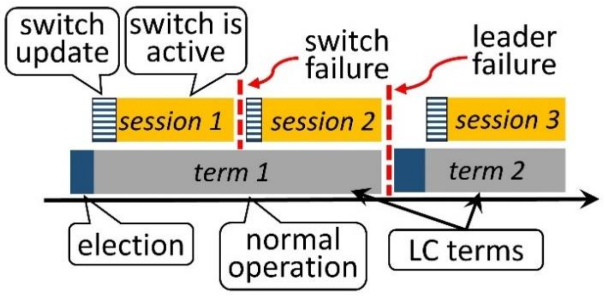


Figure 13. FLAIR sessions. Time is divided into terms. Each term starts with a leader election. Each term has one or more sessions that start with updating the switch data.

Clients. FLAIR is accessed through a client library with a simple read/write/delete interface. Read (get) and write (put) read or write entire objects. The library adds a special FLAIR packet header to every request, that contains an operation code (e.g., read) and a key (a hash-based object identifier).

Controller. Our design targets data centers that use an SDN network following a variant of the multi-rooted tree topology [44, 45]. A central controller uses OpenFlow [46] to manage the network by installing per-flow forwarding, filtering, and rewriting rules in switches.

As with previous projects that leverage SDN capabilities [47, 48, 49, 50], the controller assigns a distinct address for each replica set. The controller installs forwarding rules to guarantee that every client request for a range of keys served by a single replica set is passed through a specific switch (dubbed FLAIR switch); that switch will run the FLAIR logic for that range of keys. The controller typically selects a common ancestor switch of all replicas and installs rules to forward the system replies through the same switch. Only client requests/replies are routed through the FLAIR switch, leader-follower messages do not have the FLAIR header nor are necessarily routed through the FLAIR switch.

While this approach may create a longer path than traditional forwarding, the effect of this change is minimal. Li et al. [47] reported that for 88% of cases, there is no additional latency, and the 99th percentile had less than 5 μs of added latency. This minimal added latency is due to the fact that the selected switch is the common ancestor of target replicas and client packets have to traverse that switch anyway.

On a switch failure, the controller selects a new switch and updates all the forwarding rules accordingly. The controller load balances the work across switches by assigning different replica sets to different switches.

Storage Nodes. The storage nodes run the FLAIR and LC protocols. For read requests, before serving a read, followers verify that all committed writes to the requested object have been applied to the follower's local storage.

Write requests are processed by the leader. After a successful write operation, the leader passes to the *lflair* module the log index at which the write was committed and the list of followers that accepted the write operation and have a consistent log up to that log index. The *lflair* encodes this list into a compact bitmap and uploads it and the log index to the switch (piggybacked on the write reply).

Programmable Switch. The switch is a core component of FLAIR: it tracks every write request and the corresponding reply to identify which objects are stable (not being modified) and which replicas have a consistent value of each object (encoded in the bitmap provided by the *lfair* module). If a read is issued while there are outstanding writes for the target object (i.e., writes without corresponding replies), the read is forwarded to the leader. If a read request is processed by the switch when there are no outstanding writes to the requested object, the switch forwards the request to one of the followers included in the last bitmap for the object sent by the *lflair* module. Followers optimistically serve read requests. The switch inspects every read reply; if it suspects that a follower returned stale data (Section 4.3.6), it will conservatively drop the reply and forward the request to the leader. FLAIR forwards all writes to the leader.

FLAIR also includes techniques to handle multiple concurrent writes to the same object (Section 4.3.3), packets reordering (Section 4.3.6) and tolerating switch, node, and network failures (Section 4.3.6).

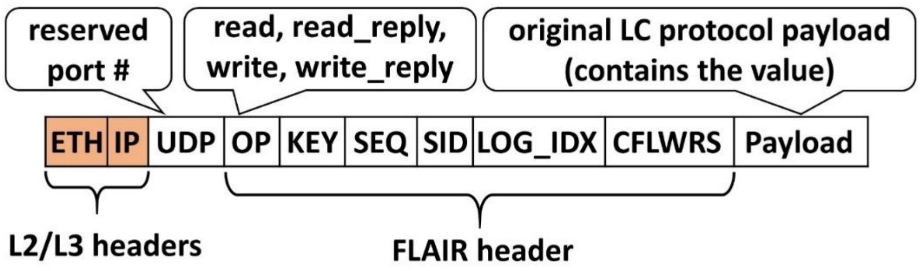

reserved port #
read, read_reply, write, write_reply
original LC protocol payload (contains the value)
ETH
IP
UDP
OP
KEY
SEQ
SID
LOG_IDX
CFLWRS
Payload
L2/L3 headers
FLAIR header

Figure 14. FLAIR packet format.

4.3 FLAIR DESIGN

4.3.1 Network Protocol

Packet format. FLAIR introduces an application-layer protocol embedded in the L4 payload of packets. Similar to many other storage systems [47, 49, 50], FLAIR uses UDP to issue client requests in order to achieve low latency and simplify request routing. Communication between replicas uses TCP for its reliability. A special UDP port is reserved to distinguish FLAIR packets; for UDP packets with this port, the switch invokes the FLAIR custom processing pipeline. Other switches do not need to understand the FLAIR header and will treat FLAIR packets as normal packets. In this way, FLAIR can coexist with other network protocols.

Figure 14 shows the main fields in the FLAIR header. We briefly discuss the fields here (a detailed discussion of the protocol is presented next):

- OP: the request type. Clients populate this field in the request packet (e.g., read, or write); replicas populate this field in the reply packets (e.g., read_reply, write_reply).
- KEY: hash-based object identifier.
- SEQ: a sequence number added by the switch. The switch increments the sequence number on every write operation.
- SID: a unique session id. The <SID, SEQ> combination represents a unique identifier for every write request.
- LOG_IDX: a log index. In a write_reply, the log index indicates the index at which the write was committed. For reads, the switch populates LOG_IDX to make sure the followers' logs are committed and applied up to that index.
- CFLWRS: In write_reply, the CFLWRS is a map of the followers that have a consistent log up to LOG_IDX.

Table 1 Session and kgroup entries. The numbers indicate the field size in bits.	

SessionArrayEntry {
bit<1> is_active;
bit<32> session_id;
bit<32> leader_ip;
bit<64> session_seq_num;
bit<48> heartbeat_tstamp; }
	KGroupArrayEntry {
	bit<1> is_stable;
	bit<64> seq_num;
	bit<64> log_idx;
	bit<8> consistent_followers;
	}

Following the FLAIR header is the original LC protocol payload, which includes the value for read/write operations.

4.3.2 Switch Data Structures

To process a read request, the switch performs two specific tasks (Section 4.3.4). First, it forwards read requests to consistent followers while balancing the load among them. Second, it verifies the read replies to preserve safety. To perform these tasks, the switch maintains two data structures: a session array and a key group array.

Session array. A single switch typically supports multiple replica sets (i.e., FLAIR+LC instances) with each set storing a disjoint set of keys. Each entry in the session array maintains the session status for a single replica set. An entry contains an is_active flag, session id, leader IP address, current session sequence number, and the timestamp of the last heartbeat received from the *lflair* module (Table 1). When is_active is true, we say the session is *active*, which indicates that the session entry and kgroup array are consistent with the leader's information. The switch processes packets using the FLAIR custom pipeline only if the session is active; otherwise, it will drop all FLAIR packets, rendering the system unavailable to clients until the switch can reach the *lflair* module and sync its session entry and key group array.

Key group (KGroup) array. To decide if followers can serve a certain read request, the switch needs to maintain information about which followers have the latest committed value of every object. Maintaining such information in the switch ASIC's memory is not feasible; instead, FLAIR groups objects based on their key and maintains aggregate information per group. We use the most significant *k* bits of the key to map an object to a key group (kgroup).

Every FLAIR+LC instance has a dedicated kgroup array. Each entry in the array (Table 1) contains the status of a single kgroup, including an is_stable flag that indicates if all objects in the kgroup are stable. If a kgroup is not stable (is_stable is false), this indicates that at least one object in the kgroup is being modified (i.e., has an outstanding write in the system). The array entry also includes the sequence number (seq_num) of the last write request processed by the switch for any object in the kgroup, the log index (log_idx) of the last successful write to any object in the kgroup, and the consistent_followers bitmap pointing to all followers that have a consistent log up to log_idx.

4.3.3 Handling Write Requests

To issue a write request, a client populates the OP and KEY fields of the FLAIR packet header and puts the value in the payload, then sends the request.

When the switch receives the request, it will mark the corresponding kgroup entry as unstable. The switch will increment the session_seq_num in the session array and use it to populate the sequence number (seq_num) in the kgroup entry and the sequence number (SEQ) in the request header. Finally, the switch populates the session id (SID) field in the header and forwards the request to the leader.

The *lflair* module will verify that the session id is valid, and will pass the write request to the leader. The leader verifies that the <SID, SEQ> combination is larger than the <SID, SEQ> number of any previous write request it ever received, else it will drop the packet. The LC leader will process the write request following the LC protocol: it will replicate the request to all followers, and when a majority of followers acknowledge the operation, the write operation is considered committed. A follower will acknowledge a write operation only if its log is identical to the leader's log up to that entry.

For the write reply, the leader will pass the following to the *lflair* module: the LC protocol payload for the write_reply, the log index at which the write was committed, and the list of followers that acknowledged the write. The *lflair* module will create the write reply packet with the leader provided payload and will populate the LOG_IDX and the bitmap of the consistent followers (CFLWRS) using the information provided by the leader. *lflair* module populates the sequence number (SEQ) in the write_reply header using the SEQ of the corresponding write request. The *lflair* module then sends the write_reply packet.

The switch will process the write_reply header and verify its session id. The switch will compare the sequence number (SEQ) of the reply to the sequence number (seq_num) in the kgroup entry; if they are equal, this signifies that no other write is concurrently being processed in the system for any object in the kgroup. Consequently, it will update the log_idx and the consistent_followers fields in the kgroup entry using the values in the write reply. Then it will mark the kgroup stable and forward the reply to the client.

If the sequence number in the reply is smaller than the sequence number in the kgroup entry, this indicates that a later write to an object in the same kgroup has been processed by the switch. In this case, the switch forwards the write reply to the client without modifying the kgroup entry. The kgroup entry remains unstable until the last write to the kgroup (with a SEQ number in the write_reply equal to the seq_num in the kgroup entry) is acknowledged by the leader.

In a nutshell, the switch acts as a look-through metadata cache. Write requests invalidate the switch metadata related to the accessed kgroup, and write replies update the kgroup metadata at the switch. As we see next, the kgroup metadata is used to consistently load balance reads.

4.3.4 Handling Read Requests

Clients fill the OP and KEY fields of the FLAIR header and send the request. When the switch receives the request, it will check the kgroup entry. If the entry is stable, the switch will fill the sequence number (SEQ) and log index (LOG_IDX) header fields using the values in the kgroup entry. Then it will forward the request to one of the followers indicated in the consistent_followers bitmap. Section 4.4.2 details our load balancing techniques.

If the kgroup entry is not stable, the switch forwards the read request to the leader. We note that there is a chance for false positives in this design, as a single write will render all the objects in the same kgroup unstable. This is a drawback of maintaining information per group of keys. This inefficiency is incurred by leases-based protocols as well, as they maintain a lease per group of objects.

When a follower receives a read request, the follower's FLAIR module validates the request, then calls advance_then_read(LOG_IDX, key) routine, which compares the follower's commit_index to LOG_IDX. If the commit_index is smaller, the follower advances its commit_index to equal LOG_IDX, apply all the log entries to the local store, then serve the read request. The FLAIR module will populate the read_reply header; for the SEQ and SID fields, it will use the values found in the read request header.

We note that it is safe to advance the follower's commit_index to match the LOG_IDX in the read request, as the switch forwards read requests to a follower only if the leader indicates that all entries in the log up to that log index are committed and that this specific follower is one of the replicas that have a log consistent to the leader's log up to that index.

When the switch receives a read_reply from a follower, it validates the session id, then verifies that the SEQ number of the read_reply equals the seq_num of the kgroup entry. If the sequence numbers are not equal, this signifies that a later write request was processed by the switch and there is a chance the follower has returned stale value. In this case, the switch drops the read_reply, generates a new read request using the KEY field from read_reply packet, and submits the read request to the leader. If the sequence number of the read_reply equals the sequance number in the kgroup entry, the switch forwards the reply to the client.

If a read request is forwarded to the leader, the *lflair* module verifies the session id, then calls advance_then_read(LOG_IDX, key). The switch verifies that the leader reply is valid (i.e., has the correct session id) before forwarding it to the client, without checking the seq_num in the kgroup entry.

4.3.5 Session Start Process

On the start of a new session, the *lflair* module reads the last session id from the LC log, increments it, and commits the new session id to the LC log. Then the *lflair* module asks the central controller for a new switch. The central controller neutralizes the old switch (making it drop all FLAIR packets) and reroutes FLAIR packets to a new switch, then confirms the switch change to the *lflair* module. This step guarantees that at any time at most one FLAIR switch is active. The *lflair* module updates the session entry (Table 1) at the switch with the current leader IP and session id. For each new session, session_seq_num is reset to zero.

Populating the kgroup array. The *lflair* module maintains a copy of the kgroup array similar to the one maintained by the switch. If the leader did not change between sessions (e.g., the session change is due to switch failure), the kgroup array at the *lflair* module is up to date. The *lflair* module will set the seq_num entry in all kgroup entries to zero (equal to the session_seq_num in the session entry)., and upload it to the switch.

If the kgroup array at the *lflair* module is empty – for instance, after electing a new leader – the *lflair* module will query the leader for three pieces of information: its commit_index, the list of followers with the same commit_index, and a list of all uncommitted operations in the log (i.e., the operations after the commit_index in the log). The list of uncommitted operations is typically small, as it only includes operations that were received before the end of the last term but were not committed yet. The *lflair* module will traverse the list of uncommitted writes and mark their target kgroup entries unstable. For all other kgroup entries, the *lflair* module will mark them stable and set their seq_num to zero, log_idx to the leader's commit_index, and consistent_followers to include all the followers that have the same commit_index as the leader's. After updating the session entry and the kgroup array at the switch, the *lflair* module activates the switch session (sets is_active to true).

4.3.6 Fault Tolerance

Follower Failure. We rely on the LC protocol to handle follower failures. To avoid sending read requests to a failing follower, the leader notifies the *lflair* module when it detects the failure of a follower. The *lflair* module removes the follower from the switch-forwarding table (Section 4.3.6).

Leader Failure. On leader failure, a new leader is elected and a new term starts. The new leader informs the *lflair* module of the term change; and the *lflair* module starts a new session.

The *lflair* module sends periodic heartbeats to the switch. Upon receiving a heartbeat, the switch determines whether it is from the current session. If the heartbeat is valid, the switch updates the heartbeat_timestamp in the session array and replies to the *lflair* module.

Switch Failure. If the *lflair* module misses the switch heartbeats for a *switch_stepdown* period of time (3 heartbeats in our prototype), the *lflair* module will suspect that the switch has failed and will start a new session. For efficiency (i.e. does not affect safety), if the switch misses three heartbeats from the leader, it will deactivate the session.

Network Partitioning. If a network partition isolates the switch from the leader, the leader treats it as a failed switch, as detailed above. If a network partition isolates the switch from a follower, read requests forwarded to the follower will time out and the client will resubmit the request. This failure affects performance, but not correctness. Upon determining that a follower is not reachable, the leader removes it from the forwarding table, as in the case of the failed follower described above.

Packet Loss. If a read or write request is lost, the client times out and resubmits the request. If a write reply is lost before reaching the switch, the kgroup entry will remain unstable until a new write operation to any key in the kgroup succeeds. While the kgroup entry is not stable, all read requests are forwarded to the leader.

Packet Reordering. It is critical for FLAIR correctness that the leader processes write requests in the same order that they are processed by the switch. Every write operation gets a unique <SID, SEQ> number. The switch marks a kgroup entry unstable until the leader replies to the last write issued for a key in the kgroup. Consequently, if the leader processes the requests out of order, the switch will incorrectly mark a kgroup stable while the out-of-order writes are modifying its objects. To prevent this scenario, the leader keeps track of the largest <SID, SEQ> it has ever processed and drops any write request with a smaller number. While session numbers (SIDs) are maintained in the log, the largest processed sequence number is retained in memory. If the leader fails, the new leader starts a new session, increments the session id (SID), and sets the session sequence number (SEQ) to zero.

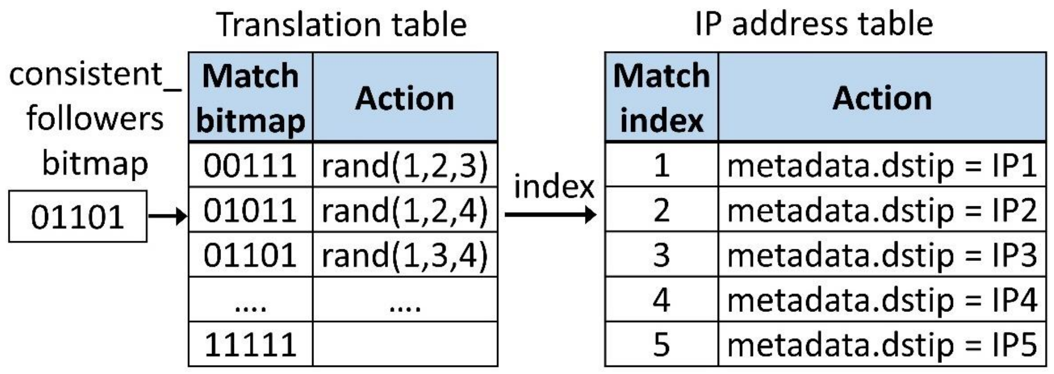


Figure 15. Logical view of the forwarding logic. The stability bitmap matches an entry in the translation table and executes the corresponding action, generating an index of the selected destination's IP address. Using the index, the IP address table sets the destination's IP address in the metadata.

4.4 FLAIR Implementation

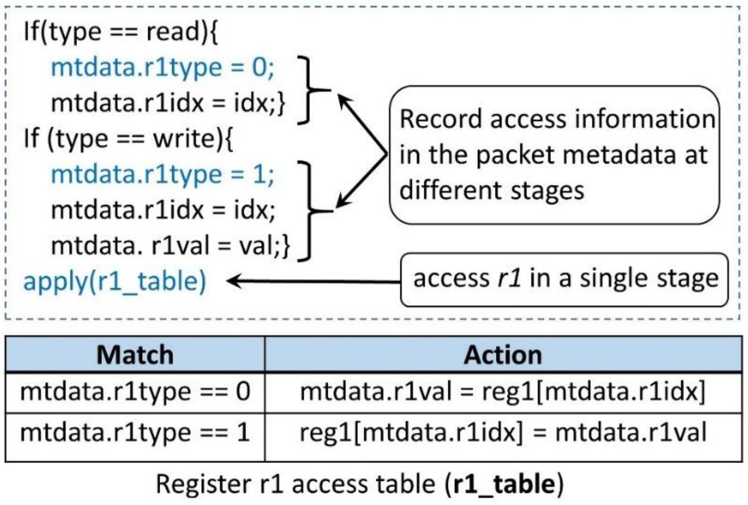
To demonstrate the benefits of the new approach, we prototyped FlairKV, a FLAIR-based key-value store built atop Raft [39]. We chose Raft due to its adoption in production systems [51, 52, 53, 54, 55], and the availability of standalone production-quality implementations [56].

4.4.1 Storage System Implementation

We have implemented FlairKV, including all switch data plane features, the FLAIR module, leaders' and followers' modifications, and the client library. We extended the Raft's follower code to implement an `advance_then_read`() function. We extended the leader to notify the *lflair* module as soon as it gets elected, and to extract its commit_index, the list of followers with a commit_index equal to the leader's commit_index, and the list of uncommitted writes. We extended the write reply with the list of followers which acknowledged the write. We implemented the leader lease optimization [7, 41] and modified Raft's client library to add the FLAIR header to client requests.

4.4.2 Switch Data Plane Implementation

The switch data plane is written in P4 v14 [40] and is compiled for Barefoot's Tofino ASIC [10], with Barefoot's P4Studio software suite [57]. Our P4 code defines 30 tables and 12 registers: six for the session array and six for the kgroup array. The kgroup array has 4K entries. Larger number of kgroups had negligible effect on performance. In total, our implementation uses less than 5% of the on-chip memory available in the Tofino ASIC, leaving ample resources to support other switch functionalities



```
If(type == read){
   mtdata.r1type = 0;
   mtdata.r1idx = idx;}
If (type == write){
   mtdata.r1type = 1;
   mtdata.r1idx = idx;
   mtdata. r1val = val;}
apply(r1_table)
```

Record access information in the packet metadata at different stages
access *r1* in a single stage

Match	Action
mtdata.r1type == 0	mtdata.r1val = reg1[mtdata.r1idx]
mtdata.r1type == 1	reg1[mtdata.r1idx] = mtdata.r1val

Register r1 access table (**r1_table**)

Figure 16. Register access table. P4 code aggregates access information that is used by a dedicated register access table.

or more FlairKV instances. The rest of this section discusses optimizations implemented in FlairKV to cope with the strict timing and memory constraints of P4 and switch ASIC.

Heartbeats implementation. The leader and the switch exchange periodic heartbeats. If the switch_stepdown period passes without receiving a leader heartbeat, the switch deactivates the session. Instead of running a process in the controller to continuously track heartbeats, the switch monitors missed heartbeats as part of the validation step in the processing pipeline. The switch keeps track of the timestamp of the last heartbeat received in the session array (Table 1). When processing any FLAIR packet, the switch computes the difference between the current time and the last heartbeat timestamp; if the difference is larger than switch_stepdown, the switch deactivates the session, making the system unavailable until the leader starts a new session.

Forwarding logic translates the consistent followers' bitmap to follower IP addresses. Storing the IP addresses of consistent followers for every entry in the kgroup array significantly increases the memory footprint. Moreover, randomly selecting a follower from the list while avoiding inconsistent ones is tricky given the P4 and current ASIC challenges (*Chapter 2*). Instead, the FlairKV leader encodes the follower status in a one-byte consistent_followers bitmap (Table 1). Replicas are ordered in a list. If the least significant bit in the consistent_follower bitmap is set, this indicates that the first replica in the list is consistent, and so forth.

When forwarding a read request, the switch translates the encoded bitmap of consistent followers to select one follower; Figure 15 shows the translation process. The consistent_followers bitmap is used as an index to the translation table. Each entry in the table has an action that randomly selects a number that is then used as an index to the IP addresses table.

This design has two benefits: it significantly reduces the memory footprint of the kgroup array, and it can be accelerated using P4 "action profiles" [58].

Load balancing. In addition to the aforementioned random load-balancing technique (Figure 15), we implemented two load-aware techniques:

- *Leader avoidance*. Our benchmarking revealed that the write operation takes 35 times longer than a read operation; most of this overhead is borne by the leader. Consequently, this load-balancing technique avoids sending read requests to the leader for stable kgroups if there are any writes in the system. The aim is to reduce the leader load, as it is already busy serving writes and serving reads for unstable kgroups.

  To implement this technique, we compare the sequence number of a write_reply with the `session_seq_num`. If they are not equal, then there are pending writes in the system and the leader should not be burdened with any reads to stable kgroups.
- *Follower load awareness*. This technique distributes the load across followers proportionally to their load in the last *n* seconds. This technique is especially useful for deployments that use heterogeneous hardware, experience workload variations, or deploy more than one replica (i.e., replicas for different ranges of keys) on the same machine.

  In our design, followers report the length of the request queue in every heartbeat. Every second, the leader calculates the average queue length for each follower and assigns proportional weights to each follower. The leader updates the translation table to reflect these weights. For instance, if follower 1 should receive double the load of any other replica, the action for a bitmap 00111 will be `rand`(1, 1, 2, 3), doubling the chance replica 1 is selected.

Register access logic. Each stage has its own dedicated registers, and a register can be accessed only once in a stage. This restriction complicates FlairKV's logic, as different packet types (e.g., read and write_reply) must access the same registers at different stages in the pipeline. To cope with this restriction, FlairKV adds a dedicated table to access each register. Figure 16 shows an example of an action table for accessing register *r1*. Our code aggregates the information about all possible modes of accessing *r1* in the packet's metadata, including the access type (read or write), the index, and which data should be written or where the value should be read to. We then use a dedicated match-action table

(Figure 16) to perform the actual read or write operation to/from the register in a single stage with a single invocation of the table. This approach has the additional benefit of reducing the number of stages.

Processing concurrent requests. The switch processes packets sequentially in a pipeline. Each pipeline stage processes one packet at a time. The switch may have multiple pipelines, each serving a subset of switch ports. FLAIR uses a single ingress pipeline and all egress pipelines. If a FLAIR packet is received on a different ingress pipeline, the packet is recirculated [58] to the FLAIR pipeline.

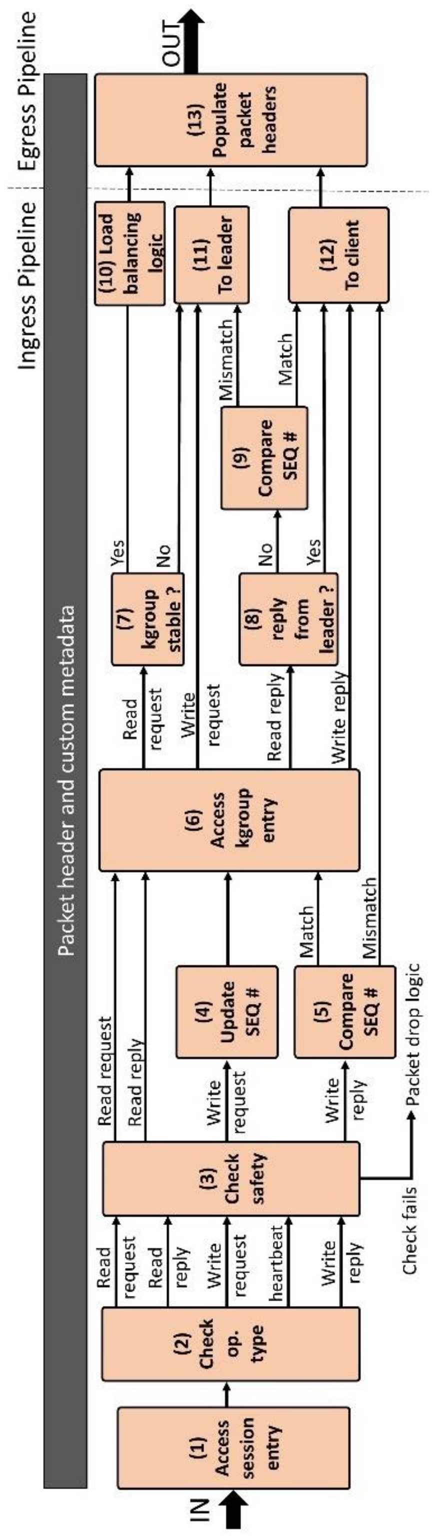


Figure 17. Logical view of the FlairKV switch data plane.

4.4.3 Putting the Switch Pipeline Together

Figure 17 shows the pipeline layout in the switch data plane and the flow for a FlairKV packet. The pipeline starts by reading the session information (1 in Figure 17) and adding it to the packet metadata. Then the it extracts the operation type (2) and validates the request (3) by verifying the session id. If the packet has an older session id the packet is dropped. Further, in the validation stage the switch confirms that it did not miss leader heartbeats in the last switch_stepdown period (Section 4.3.5), else it deactivates the session.

Read requests access the kgroup array (6), and if the group is stable, the request is forwarded to a load-balancing logic (10) that implements the forwarding logic (Section 4.4.2); otherwise, it is sent to the leader.

If a read reply is from the leader, it is forwarded to the client (12). If it is from a follower, the pipeline performs the safety check (9) and, if it suspects the reply is stale, drops the reply, then resubmits the read request to the leader (11).

Write requests update the `session_seq_num` (4) and the kgroup entry (6), then are sent to the leader (11).

Write replies compare the sequence number of the reply to the one in the kgroup entry (5); if they match, the kgroup entry is updated (6) and the pipeline forwards the reply to the client (12).

The egress pipeline (13) has one logical stage that populates the header fields (e.g., `SEQ` number, `SID`, etc.) using the data available in the packet's metadata.

4.5 FLAIR Evaluation

We compare our prototype with previous approaches in terms of throughput and latency (§4.5.1) with different workload skewness (§4.5.2) and read/write ratios (§4.5.3).

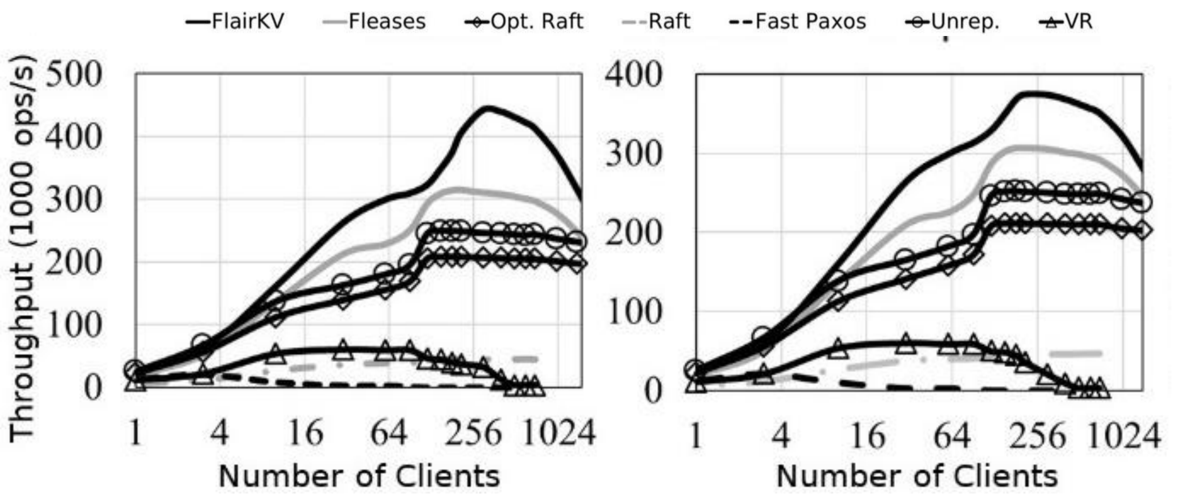
Testbed. We conducted our experiments using a 13-node cluster. Each node has an Intel Xeon Silver 10-core CPU, 48GB of RAM, and 100Gbps Mellanox NIC. The nodes are connected to an Edgecore Wedge 100 ×32BF switch with 32 100Gbps ports. The switch has Barefoot's Tofino ASIC, which is P4 programmable. Unless otherwise specified, three machines ran the server code, while the other 10 machines generated the workload.

Alternatives. We compare the throughput and latency of the following designs and optimizations:

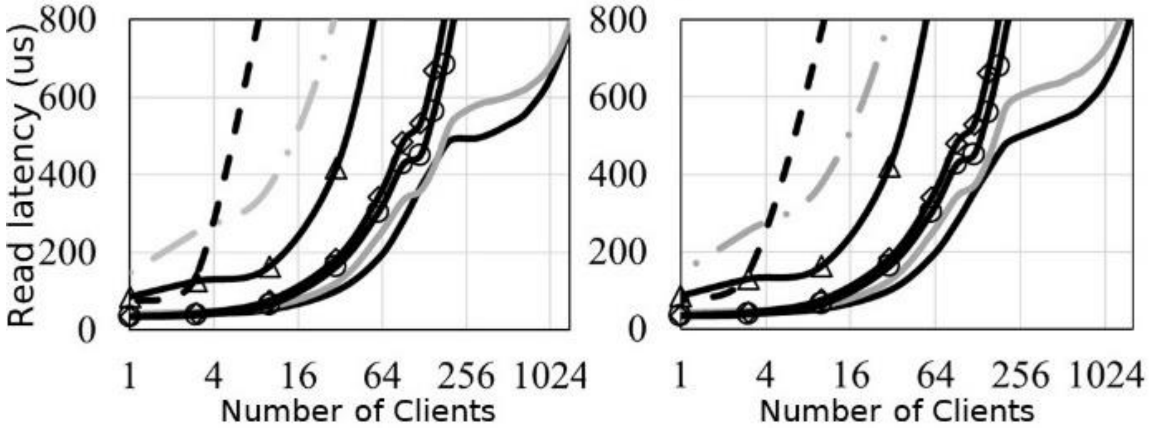
- **Leader-based**. We used two leader-based protocol implementations: LogCabin, the original implementation of Raft (**Raft**), and an implementation of Viewstamped replication (**VR**) [59]. Raft and VR implement a batching optimization which batches and replicates multiple log entries in a single round trip.

- **Optimized Leader-based (Opt. Raft).** Our benchmarking revealed that the original Raft implementation could not utilize the resources of our cluster. We implemented two main optimizations: first, we changed the request-processing logic from an event-driven to a thread-pool design, as our benchmarking indicated a thread-pool performs better; second, we implemented the leader-lease optimization. These changes significantly improved Raft's performance.
- **Quorum-based reads (Fast Paxos).** An alternative to the leader-based design is the quorum design [47, 48, 60]. Typically, client read requests are sent to all followers, and each follower responds directly to the client. The client waits for a reply from a supermajority [60] before completing a read. We used a Fast Paxos implementation that implements only the normal case [59].
- **Follower-lease optimization (FLeases).** Similar to MegaStore [8], the leader grants read leases to all followers. Before serving a write, the leader revokes all leases, processes the write operation, and then grants a new lease to followers. The lease's grant/revoke messages are piggybacked on the consensus protocol messages. However, writes should be processed by *all* followers before replying to the client. In our experiments, if a follower receives a read request for an object for which it does not have an active lease, it forwards the request to the leader. MegaStore applications typically partition the keys into thousands of groups, each group contains logically-related keys [8] (e.g., a key group per blog [8]). We partitioned the keys into 4K groups (the same number of kgroups in FlairKV), and followers get a lease per group. Clients randomly select a follower for each read request and send the request directly to it.
- **Unreplicated/NOPaxos (Unrep.).** As a baseline, the unreplicated configuration deploys Optimized-Raft (discussed above) on a single node. The single node stores the data set and serves all operations without replication.

  This configuration also represents the best possible performance of the network-optimized NOPaxos [47] protocol. NOPaxos uses a network switch to order and multicast read and write operations to all replicas. An operation is successful if the majority accepts a write or returns the same value for a read. Consequently, NOPaxos read performance is limited by the slowest node in the majority of nodes. NOPaxos evaluation shows that the best throughput and latency the protocol can achieve are within 4% that of an unreplicated system [47].



(a) Throughput – Uniform (b) Throughput -Zipf



(c) Latency-Uniform (d) Latency-Zipf

Figure 18. Throughput and Latency while varying the number of clients. The figures show the throughput and the average latency for different number of clients for workload B for the uniform distribution (a, c), and for the Zipf distribution (b, d).

- **FlairKV.** Unless otherwise specified, we used FlairKV with the leader-avoidance load-balancing technique.

We benchmarked every system and selected a configuration that maximized its performance. We stored all data in memory. In all experiments, all systems' performance (with the exception of FastPaxos) was stable with a standard deviation less than 1%.

Workload. We used synthetic benchmarks and the YCSB benchmark **[61]** to evaluate the performance of all systems. In our evaluation, we considered both uniform and skewed workloads. The skewed workload follows the Zipf distribution with a skewness parameter of 0.99. We also used the YCSB benchmark. We experimented with 100,000 and 1 million keys. We present the results with 100,000 keys as, in skewed workloads, the fewer number of hot keys increased the chance of having concurrent requests accessing the same key (i.e. is less favorable for FlairKV). FlairKV brings slightly higher performance benefit when using 1 million keys than 100,000 keys. The key size is 24 bytes and the hash of the key string is used as the key in the FLAIR protocol. The value size is 1KB.

4.5.1 Performance Evaluation

We compared the seven systems using YCSB workload B (95:5 read:write ratio) while varying the number of clients, with uniform and skewed workload distribution. Figure 18 shows the throughput and average latency with a uniform and skewed distributions. With the uniform distribution (Figure 18 (a) and (c)), FlairKV achieves up to 42% higher throughput and 23.7% lower average latency than FLeases, and 1.3 to 2.1 times higher throughput and 1.5 to 2.4 times lower latency compared to optimized Raft and unreplicated setup. Fast Paxos, Raft, and VR, achieve the lowest throughput and highest latency as these systems contact the majority of nodes for every read.

FlairKV achieved better performance than FLeases for three reasons. First, FlairKV uses the leader-avoidance load-balancing technique, which reduces the load on the leader when there are writes, thereby accelerating writes and shortening the time period in which kgroups are marked unstable. This approach is effective as writes take almost 35 times longer than reads in Opt.Raft, and 30 times longer in the unreplicated setup. We recorded the number of read requests served by the leader. For instance, with 300 clients (Figure 18.a) the leader served 2% of the reads in FlairKV (those are reads to unstable kgroups), while it served 34% of the reads in FLeases. We note that the leader-avoidance technique cannot be applied to FLeases which tasks the clients with selecting a follower to send the read request to. This technique requires accurate information about the current load of the leader and which followers are stable which are not available to clients.

Second, in FLeases, when an object is not stable, if a client sends a request to a follower, the follower will redirect the request to the leader, increasing overhead and incurring extra latency. Unlike FLeases, FlairKV switch knows if an object is not stable and forwards read requests for that object directly to the leader. The third reason which had a minor impact when using 3 replicas is that the write operation in FLeases need to reach all followers, while FlairKV writes only need a majority.

Optimized-Raft's performance is better than that of Raft, VR, and FastPaxos. The unreplicated deployment slightly improves throughput and latency over Optimized-Raft by avoiding the replication overhead for write operations. These two systems still lag behind FlairKV as they only utilize a single node (the leader) for serving all reads and writes.

Figure 18 (b) and (d) show the throughput and average latency with a skewed workload (Zifpian constant of 0.99). The skewed workload results in higher contention and an increased frequency at which a read request finds a kgroup unstable. This contention reduces the chances of reading from followers. FlairKV leader served 21% of reads of which 1% are redirected from followers, while FLeases leader served 37% of reads. Even under the skewed workload, FlairKV still achieves the highest performance, up to 26% higher throughput and 18.1% lower latency than FLeases, and 1.5 to 1.8 times higher throughput and 2 to 2.4 times lower latency than optimized Raft and the unreplicated setup.

Latency evaluation. Figure 21.a shows the latency CDF of FlairKV, FLeases, OptRaft, and Raft. Under the uniform workload B with 300 clients (other workloads had similar results). FlairKV lowered the latency for the slowest 40% requests by at least 38% relative to FLeases. Under the Zipf workload (Figure 21.b), FlairKV lowered the slowest 50% of request by up to 35% relative to FLeases.

FLeases has higher latency as it incurs extra delay due to the load imbalance between nodes (e.g., the leader serves 41% of requests for workload B with Zipf distribution) and due to followers redirecting 4% of requests to the leader.

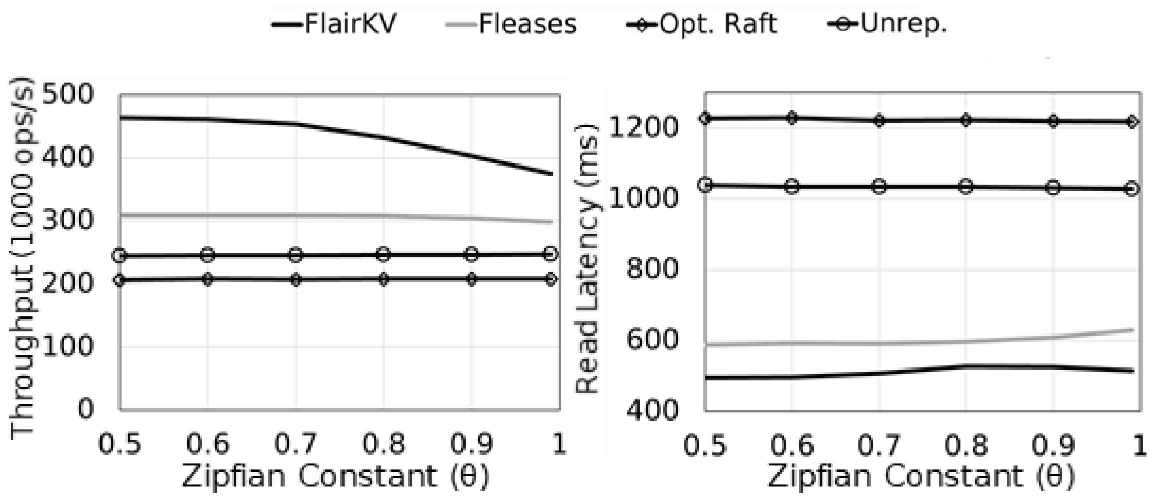
Under all workloads, FlairKV significantly improved operation's latency relative to OptRaft and Raft. The median latency of FlairKV is 2% of Raft's latency and 2-8% of OptRaft's latency.

4.5.2 Workload Skewness

We measured the impact of the workload skewness on throughput (Figure 19.a) and average latency (Figure 19.b) by varying the Zipfian constant from 0.5 to 0.99. FlairKV consistently achieves better performance: 1.26 to 2.25 times higher throughput and 1.13 to 2.48 times lower average latency compared to all other systems. We notice that as the skewness increases FlairKV and FLeases performance decreases as higher skewness increases contention on the few popular kgroups, making them unstable for longer time, and increases the number of requests the leaders have to process. Other systems performance is not noticeably affected by skewness.

We noticed high workload skewness affects FlairKV's performance more than FLeases. This is due to a subtle side effect of FlairKV. When there are concurrent writes to the same kgroup, FlairKV will mark a group unstable from the moment the first request is processed by the switch until the last request to the kgroup is replied to ([*t1*, *t2*] in Figure 22). In FLeases, the lease revocation is piggybacked on the write replication step (black diamonds in Figure 22). Once the leader commits a write, it sends a commit notification and grants a new lease to the followers (white diamonds). Hence, FLeases may grant a lease between concurrent writes, creating more opportunity for serving reads from followers.

To further understand this effect, we tracked leases and the stability of kgroups under the skewed (factor of 0.99) write heavy YCSB workload A (1:1 read:write ratio). We noticed that while 29% of reads found the kgroup unstable in FlairKV, only 4% of reads in FLeases reached a follower that did not have a lease. We further profiled the write operation path and found that FLeases revokes leases for 75% of the write operation time (Figure 22) 25% shorter than the period FlairKV marks a kgroup unstable. Despite this subtle effect FlairKV leader still has lighter load, it served 29% of reads compared to 37% served by the FLeases leader. Notwithstanding this effect FlairKV still brings 17% to 26% performance improvement even under skewed workloads.



(a) Throughput (b) Read latency

Figure 19. Throughput and Latency while varying skewness. The figures show the throughput (a) and the average latency (b) for different zifpian constants for a uniform workload B with 300 clients.

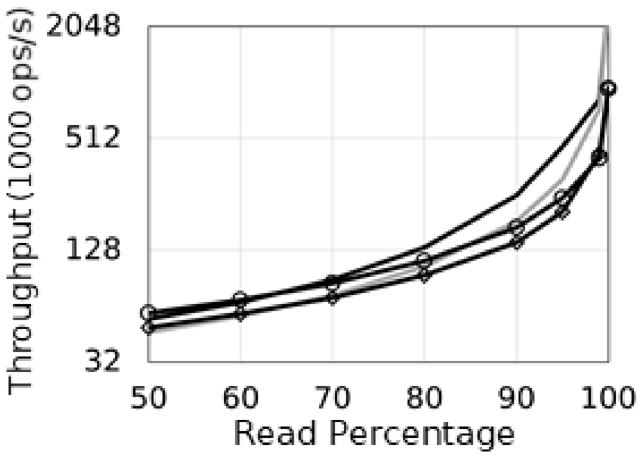
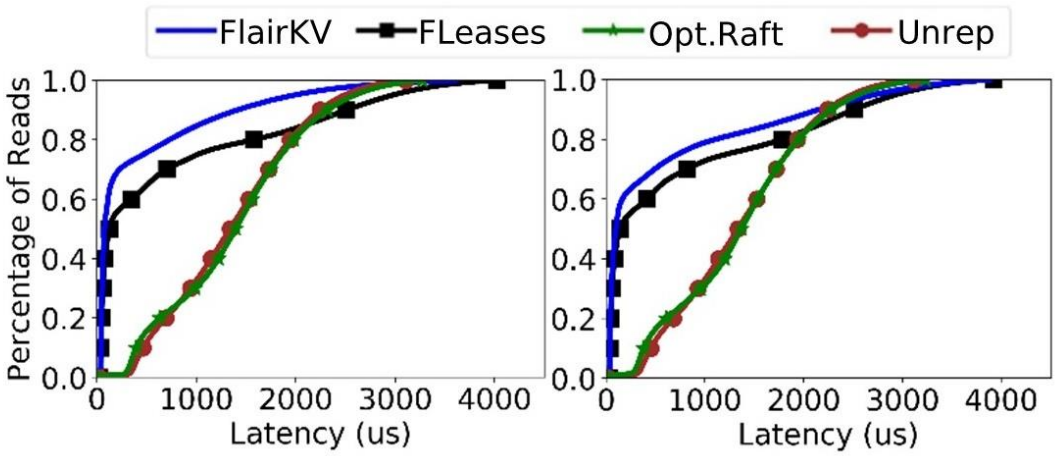


Figure 20. Throughput while varying the read ratio. Using uniform workload B.

4.5.3 Read/Write Ratio

Figure 20 shows the effect of the ratio of reads to writes on systems' performance with a uniform workload B. Compared to FLeases, FlairKV has up to 1.5 times higher throughput for all read to write ratios, with the exception of the read-only workload in which their performance is comparable. FlairKV has 1.25 to 2.8 times higher throughput compared to the Opt. Raft. Compared to the unreplicated setup, FlairKV has up to 2.8 times higher throughput for workloads with 70% reads or more and a comparable performance under write heavy workloads (read ratio 50-70%).



(a) B-Uniform (b) B-Zipf

Figure 21. Latency CDF. The figures show the latency CDF for reads under workload B using 300 clients with a uniform distribution (a), and a Zipf distribution with skewness of 0.99 (b). The lines for Opt. Raft and Unrep. almost overlap.

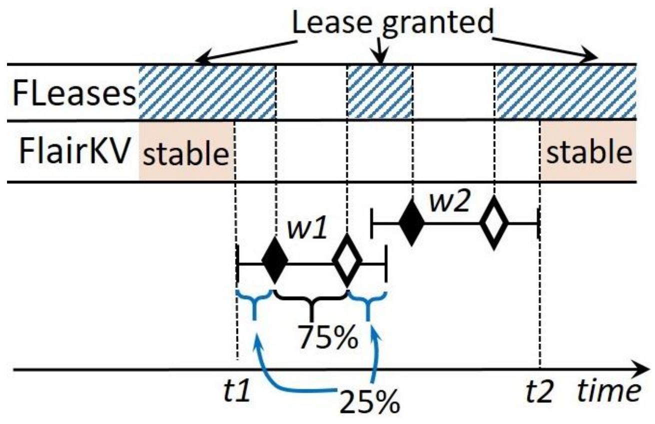


Figure 22. Subtle effect of FLAIR. FLeases may grant leases for up to 25% more time compared to FlairKV. Bars mark the time from the moment a switch receives a write request (w1 or w2) until it receives a corresponding reply.

4.5.4 Fault Tolerance

To demonstrate FlairKV fault tolerance techniques, we measured the system throughput using workload C under three failure scenarios: switch, leader, and follower failure.

Switch Failure. We ran FlairKV at peak throughput for 35 seconds (Figure 23). At the 10s mark, the controller emulated a switch failure by wiping out the switch registers and installing rules to drop switch heartbeats. After missing 3 heartbeats, the leader suspects that the switch has failed and starts a new

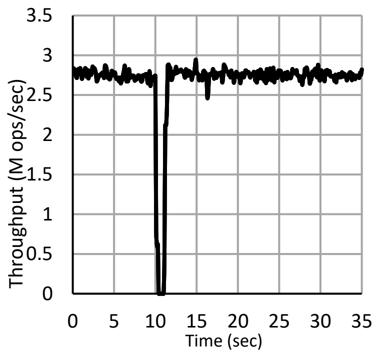


Figure 23. FlairKV throughput during a switch failover.

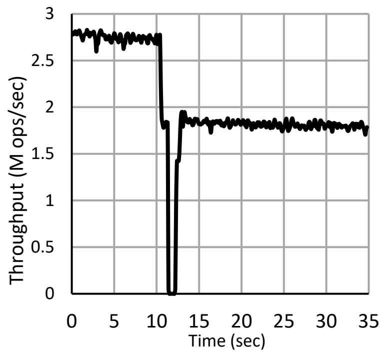


Figure 24. FlairKV throughput during leader failover.

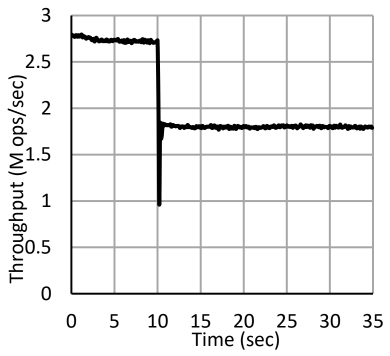


Figure 25. FlairKV throughput during a follower failure.

session. During this process, the switch is inactive, which causes the throughput to drop to zero for 750ms. Afterwards, the switch resumes normal operations.

Leader Failure. Figure 24 shows FlairKV throughput during the leader failure. We ran FlairKV at peak throughput for 35 seconds. At the 10s mark, we kill the leader process. Write requests fail, but the switch continues to forward read requests to followers. After missing 3 heartbeats the switch deactivates the session, and the throughput drops to zero. After 6 heartbeats, the followers elect a new leader that starts a new session. The system resumes its operation with one leader and one follower.

Follower Failure. We ran FlairKV at peak throughput for 35 seconds (Figure 25). At the 10s mark, we kill a follower process. This causes a drop in throughput as fewer replicas are available to serve read requests. The switch keeps forwarding client requests to the failed follower until the leader updates the switch. The dip in throughput at the second 10 is because we use closed-loop clients and some of the clients block waiting for the failed replica before timing out and retrying. Afterwards, the system throughput drops by 33% due to the loss of one follower.

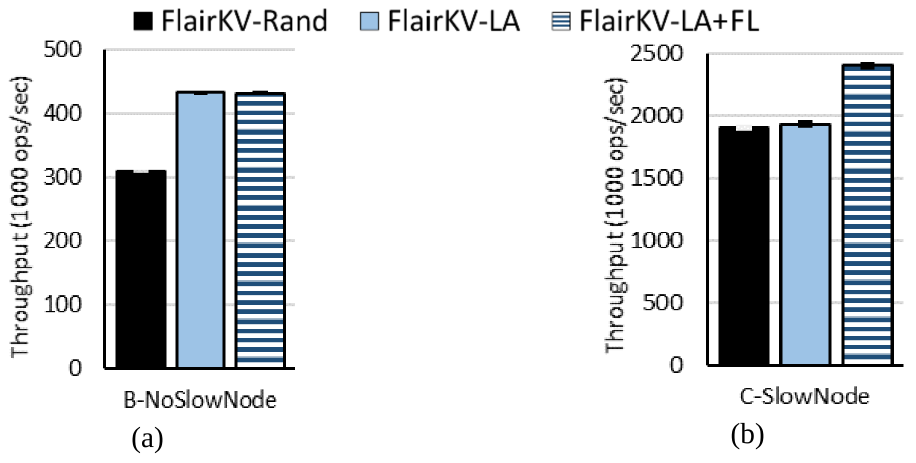


Figure 26. Throughput using different load-balancing techniques. (a) Uses workload B without slowing any follower and (b) uses workload C and slows one of the followers.

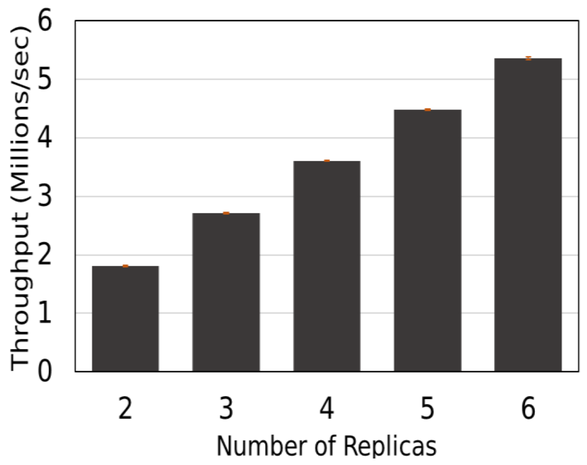


Figure 27. FlairKV scalability with different number of replicas

4.5.5 Scalability

To demonstrate FlairKV scalability, we measured the system throughput using a read-only YSCB workload C while varying the number of replicas (Figure 27). The figure shows that FlairKV throughput scales linearly with the number of replicas, reaching 5.4 million request per second with 6 followers. We notice that the system achieves much higher performance under the read-only workload mainly due to the lower operation overhead (as writes take 30 times longer even without accounting for the replication overhead).

4.5.6 Load-balancing Performance Evaluation

We measured the system throughput using the following three configurations of FlairKV (detailed in 4.4.2)

- *FlairKV-Rand* selects a follower or the leader at random. Consequently, read requests for stable kgroups are uniformly spread across the followers and the leader.
- *FlairKV-LA* applies the leader-avoidance technique.
- *FlairKV-LA+FL* uses both leader-avoidance and follower load-awareness techniques.

Figure 26.a shows the performance improvement produced by the leader-avoidance technique. In this experiment, we used workload B with uniform key popularity distribution. The results show that FlairKV-LA throughput is higher by 40% than FlairKV-Rand throughput, as it accelerates writes and reduces the period in which kgroups are marked unstable. FlairKV-LA+FL had comparable performance to FlairKV-LA as nodes are homogenous.

Figure 26.b evaluates the benefits of using the follower-load awareness technique (4.4.2). This technique helps in deployments with heterogeneous hardware and with load variance. To emulate such scenarios, we manually reduced the CPU frequency for one follower by 10%. We used the read-only workload C with uniform distribution to avoid write operations (as those give advantage to the leader-avoidance technique). FlairKV-LA+FL had 17% higher throughput relative to the other configurations, as it distributes the load proportionally to the node's request queue length. Furthermore, we noticed that FlairKV-LA+FL reduces latency by 10%. FlairKV-LA and FlairKV-Rand are equivalent under the read-only workload.

4.6 Summary

We present FLAIR, a novel protocol that leverages the capabilities of the new generation of programmable switches to accelerate read operations without affecting writes or using leases. FLAIR identifies, at line rate, which replicas can serve a read request consistently, and implements a set of load-balancing techniques to distribute the load across consistent replicas. We detailed our experience building FlairKV and presented a number of techniques to cope with the restrictions of the current programmable switches. We hope our experience informs a new generation of systems that co-design network protocols with system operations.

Chapter 5
Related Work

Centralized Scheduling. Spark [3] uses a centralized scheduler which works by pushing tasks to worker nodes whenever they are free to run them. However, as we have shown previously, it is inadequate for low latency tasks because centralized schedulers become the bottleneck in larger clusters.

To address these issues, Drizzle [18] and Hopper [19] proposed techniques to improve centralized scheduler performance. Drizzle [18] explored reusing scheduling decisions across jobs and leveraging data dependency between jobs to collocate dependent tasks. Hopper [19] uses speculation to reduce the effect of straggler tasks and demonstrates that this technique can be used with centralized and distributed schedulers. However, such systems are still unable to address the scalability challenge and tend to become the performance bottleneck at large cluster configurations, which Falcon avoids despite being centralized as it can make scheduling decisions at line-rate.

Distributed Scheduling. Sparrow [2], Hopper [19], and Apollo [20] present a distributed scheduling approach. Sparrow addresses the scalability issue by having multiple stateless schedulers that do not communicate with each other. To improve scheduling accuracy, Sparrow combines late-binding with the power-of-two techniques. However, for very high cluster utilization rate, the scheduling accuracy gets worse, as it becomes much more difficult to send probes to the nodes that will become free soon. Although Sparrow is the state-of-the-art low-overhead scheduler, it still adds a few milliseconds of scheduling overheads, making it unfeasable to use with tasks that take tens of millisecond to execute.

Hopper tries to incorporate speculation, which is the primary defense mechanism against stragglers, in synergy with its scheduling policy. It tries to estimate the “virtual size” of each job empirically and via real-time measurements to determine the threshold past which allocating additional resources to a job will have diminishing returns. Hopper has been implemented in both centralized and decentralized fashion with the decentralized implementation remaining very close to the centralized implementation in terms of overall performance. It does this by using late binding mechanisms similar to Sparrow but by replacing the power of two choices with the “power of many choices” since this is better suited for heavy tailed jobs which are likely to generate stragglers. Apollo [20] uses multiple schedulers that access a shared metadata service. The metadata service offers approximate cluster status. However, because the metadata is not accurate, this may lead to suboptimal scheduling decisions.

Lock-free FIFO queues. There are many proposed lock-free algorithms to implement a shared FIFO queue. Michael et al. [62] and Valios [63] proposed simple lock-free algorithms for a shared queue using compare-and-swap (CAS) primitive for both adding and retrieving tasks. Prakash et al. [64] proposed an algorithm that takes snapshots to the current queue status, this information helps threads to determine if there is any blocked thread, and completes its job. However, all these queue designs access some variable multiple times (e.g. one time to read the tail pointer, and another time to make sure that it is still valid before adding a task), which makes these designs not suitable for modern programmable switches.

Hybrid Scheduling. Finally, a few scheduling systems use a hybrid approach. Hawk [5] uses centralized scheduling for long-running tasks and distributed scheduling for low-latency tasks. It devides the cluster into two partitions responsible for short and long running tasks in order to avoid head-of-line blocking. Mercury [65] uses a hybrid model and provides an API to allow the client to choose the scheduling approach for its jobs. These distributed schedulers provide a low-latency scheduling but are based on approximate cluster information.

Network-Accelerated Systems. Recent projects have utilized programmable switches to implement in-network caching [66], DNN training and inferencing [67], in-network aggregation operations [68], provide load balancing [69, 70, 71], access control [72], seamless virtual machine migration [73], and improving system security, virtualization, and network efficiency [74]. NetCache [66] exploits the fact that programmable switches can access their memory at line-rate and built an in-network look-through caching layer for key-value stores layer. DAIET [68] uses programmable switches in order to accelerate aggregation operations. It uses a tree of programmble switches that performs aggregation operations while packets are traversing the network. This reduces the bandwidth taken to perform aggregation, and reduces the latency. SwitchKV [49] uses SDN capabilities to route client requests to the caching node serving the key. A central controller populates the forwarding rules to invalidate routes for objects that are being modified and installs routes for newly cached objects. However, none of these previous projects handled a scheduling problem for data analytics or built a pipeline-based task queue.

Network-accelerated consensus. A number of recent efforts leverage SDN's capabilities to optimize consensus protocols. Speculative Paxos [48] builds a mostly ordered multicast primitive and uses it to optimize the multi-Paxos consensus protocol. Network-ordered Paxos (NOPaxos) [47] leverages modern network capabilities to order multicast messages and add a unique sequence number to every client request. NOPaxos uses these sequence number to serialize operations and to detect packet loss. Speculative Paxos and NOPaxos are optimized for operations that update the log but not for read operations. NetChain [75] and NetPaxos [76] implement replication protocols on a group of switches. These protocols are suitable for systems that store only a few megabytes of data (e.g., 8MB in the NetChain prototype). Unlike FLAIR, these efforts do not optimize for read operations. Reads are still served by the leader or a quorum of replicas.

Consensus protocols optimized for the WAN. A number of consensus protocols are optimized for WAN deployments. Quorum leases [37] proposes giving a read lease to some of the followers; Unlike Megastore leases, when an object is modified, only the followers that have the lease are contacted. Quorum leases has a better performance than Megastore leases in WAN setups, but do not bring benefits when deployed in a single cluster [37]. Mencius [77] is a multi-leader protocol in which each leader controls part of the log. EPaxos [78] is a leaderless protocol where clients can submit request to any replica. Non-conflicting write can commit in one round trip, while conflicting writes will be resolved using Paxos.

CURP [79] optimizes the write operation through exploiting commutativity between concurrent writes. In data center deployments, CURP reads are served by the leader and hence are limited to a single node performance, in WAN deployment CURP applies a technique similar to FLeases.

Chapter 6
Conclusion

In this thesis, we exploited data-plane programmable network devices in order to accelerate scheduling in compute clusters. Offloading scheduling to the network help to improve throughput, latency, and scalability.

We presented Falcon, a centralized in-network scheduler that can assign tasks to the next available worker at line-rate and scale to process billions of requests per second. Our evaluation shows that Falcon can reduce scheduling overhead by an order of magnitude compared to current state-of-the-art low-latency schedulers.

Compared to other projects that leverage programmable switches, Falcon explores the feasibility of offloading a complete service to the network. Falcon shows that, instead of running a scheduler on one or more servers, the complete scheduling service can be offloaded to the network.

Moreover, we utilized programmable switches to perform consistency-aware load-balancing for key-value stores. We built FLAIR, a novel protocol that allows for serving and load balancing read requests among follower replicas. Our evaluation shows that FLAIR brings significant performance gains: up to 42% and lowered 35-97% of the latency compared to the current state-of-the-art approaches.

In summary, this thesis presents a new design paradigm for building accurate scheduler for large scale clusters. We demonstrate the feasibility and the benefits of this approach through two systems that target diverse application domains and workloads. Our evaluation shows the feasibility of this approach and invites further research for offloading other scheduling techniques to programmable switches.

References

[1] M. Poke and T. Hoefler, "DARE: High-Performance State Machine Replication on RDMA Networks," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, Portland, Oregon, USA, 2015, 2749267: ACM, pp. 107-118, doi: 10.1145/2749246.2749267.

[2] K. Ousterhout, P. Wendell, M. Zaharia *et al.*, "Sparrow: distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 69-84.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin *et al.*, "Spark: Cluster computing with working sets," *HotCloud,* vol. 10, no. 10-10, p. 95, 2010.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, 2004, vol. 6: USENIX Association, pp. 137--150.

[5] P. Delgado, F. Dinu, A.-M. Kermarrec *et al.*, "Hawk: Hybrid datacenter scheduling," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 499-510.

[6] "Tofino-2 Second-generation of World's fastest P4-programmable Ethernet switch ASICs." https://www.barefootnetworks.com/products/brief-tofino-2/ (accessed 16-03-2020).

[7] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the USENIX Annual Technical Conference*, Philadelphia, PA, 2014: USENIX Association.

[8] J. Baker, C. Bond, J. C. Corbett *et al.*, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011.

[9] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, Washington, 2006: USENIX Association.

[10] "Barefoot Tofino." https://www.barefootnetworks.com/products/brief-tofino/ (accessed April 14, 2019.

[11] "Cavium / XPliant." https://origin-www.marvell.com/documents/netpxrx94dcdhk8sksbp/ (accessed April 14, 2019.

[12] "High-Capacity StrataXGS® Trident 3 Ethernet Switch Series." https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series (accessed September 9, 2019.

[13] P. Bosshart, D. Daly, G. Gibb *et al.*, "P4: programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.,* vol. 44, no. 3, pp. 87-95, 2014, doi: 10.1145/2656877.2656890.

[14] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM Sigmod Record,* vol. 34, no. 4, pp. 42-47, 2005.

[15] S. Wang, J. Liagouris, R. Nishihara *et al.*, "Lineage stash: fault tolerance off the critical path," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 338-352.

[16] K. Ousterhout, A. Panda, J. Rosen *et al.*, "The case for tiny tasks in compute clusters," in *Proceedings of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[17] T. Zhang, A. Chowdhery, P. Bahl *et al.*, "The design and implementation of a wireless video surveillance system," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, 2015, pp. 426-438.

[18] S. Venkataraman, A. Panda, K. Ousterhout *et al.*, "Drizzle: Fast and adaptable stream processing at scale," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 374-389.

[19] X. Ren, G. Ananthanarayanan, A. Wierman *et al.*, "Hopper: Decentralized speculation-aware cluster scheduling at scale," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 379-392.

[20] E. Boutin, J. Ekanayake, W. Lin *et al.*, "Apollo: Scalable and coordinated scheduling for cloud-scale computing," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 285-300.

[21] S. Hendrickson, S. Sturdevant, T. Harter *et al.*, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[22] P. Bosshart, D. Daly, G. Gibb *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review,* vol. 44, no. 3, pp. 87-95, 2014.

[23] J. Li, E. Michael, N. K. Sharma *et al.*, "Just say NO to paxos overhead: Replacing consensus with network ordering," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 467-483.

[24] X. Li, R. Sethi, M. Kaminsky *et al.*, "Be fast, cheap and in control with SwitchKV," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 31-44.

[25] S. Al-Kiswany, S. Yang, A. C. Arpaci-Dusseau *et al.*, "NICE: Network-integrated cluster-efficient storage," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 29-40.

[26] P. Hunt, M. Konar, F. P. Junqueira *et al.*, "ZooKeeper: wait-free coordination for internet-scale systems," in *Proceedings of the USENIX annual technical conference*, Boston, MA, 2010.

[27] B. Calder, J. Wang, A. Ogus *et al.*, "Windows Azure Storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011, doi: 10.1145/2043556.2043571.

[28] J. C. Corbett, J. Dean, M. Epstein *et al.*, "Spanner: Google's globally-distributed database," in *Proceedings of the USENIX conference on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, USA, 2012: USENIX Association.

[29] N. Bronson, Z. Amsden, G. Cabrera *et al.*, "TAO: Facebook's distributed data store for the social graph," in *Proceedings of the USENIX Technical Conference*, San Jose, CA, 2013: USENIX Association.

[30] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, Inc., 2004.

[31] L. Lamport, "Paxos made simple," *ACM Sigact News,* vol. 32, no. 4, pp. 18-25, 2001.

[32] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems&Networks*, 2011: IEEE Computer Society, doi: 10.1109/dsn.2011.5958223.

[33] B. Liskov and J. Cowling, "Viewstamped replication revisited," Technical Report MIT-CSAIL-TR-2012-021, MIT, 2012.

[34] J. Shute, R. Vingralek, B. Samwel *et al.*, "F1: a distributed SQL database that scales," *Proc. VLDB Endow.,* vol. 6, no. 11, pp. 1068-1079, 2013, doi: 10.14778/2536222.2536232.

[35] B. Atikoglu, Y. Xu, E. Frachtenberg *et al.*, "Workload analysis of a large-scale key-value store," presented at the Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems, London, England, UK, 2012.

[36] C. Gray and D. Cheriton, "Leases: an efficient fault-tolerant mechanism for distributed file cache consistency," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 1989: ACM, doi: 10.1145/74850.74870.

[37] I. Moraru, D. G. Andersen, and M. Kaminsky, "Paxos Quorum Leases: Fast Reads Without Sacrificing Writes," in *Proceedings of the ACM Symposium on Cloud Computing*, Seattle, WA, USA, 2014: ACM, doi: 10.1145/2670979.2671001.

[38] J. Terrace and M. J. Freedman, "Object storage on CRAQ: high-throughput chain replication for read-mostly workloads," presented at the Proceedings of the 2009 conference on USENIX Annual technical conference, San Diego, California, 2009.

[39] "LogCabin storage system." https://logcabin.github.io (accessed April 14, 2019.

[40] "P4." https://p4.org (accessed April 14, 2019.

[41] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the annual ACM symposium on Principles of distributed computing*, Portland, Oregon, USA, 2007: ACM, doi: 10.1145/1281100.1281103.

[42] D. Mazieres, "Paxos made practical," ed, 2007.

[43] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.,* vol. 16, no. 2, pp. 133-169, 1998, doi: 10.1145/279227.279229.
[44] "Data Center: Load Balancing Data Center." https://learningnetwork.cisco.com/docs/DOC-3438 (accessed April 14, 2019.
[45] L. A. Barroso and U. Hoelzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009, p. 120.
[46] N. McKeown, T. Anderson, H. Balakrishnan *et al.*, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.,* vol. 38, no. 2, pp. 69-74, 2008, doi: 10.1145/1355734.1355746.
[47] J. Li, E. Michael, N. K. Sharma *et al.*, "Just say no to paxos overhead: replacing consensus with network ordering," in *Proceedings of the USENIX conference on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, USA, 2016: USENIX Association.
[48] D. R. K. Ports, J. Li, V. Liu *et al.*, "Designing distributed systems using approximate synchrony in data center networks," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Oakland, CA, 2015: USENIX Association.
[49] X. Li, R. Sethi, M. Kaminsky *et al.*, "Be fast, cheap and in control with SwitchKV," in *Proceedings of the Usenix Conference on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, 2016: USENIX Association.
[50] S. Al-Kiswany, S. Yang, A. C. Arpaci-Dusseau *et al.*, "NICE: Network-Integrated Cluster-Efficient Storage," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, Washington, DC, USA, 2017: ACM, doi: 10.1145/3078597.3078612.
[51] "etcd: Distributed reliable key-value store for the most critical data of a distributed system." https://github.com/etcd-io/etcd (accessed April 14, 2019.
[52] "RethinkDB: the open-source database for the realtime web." https://www.rethinkdb.com/ (accessed April 14, 2019.
[53] "Open Network Operating System (ONOS) - Cluster Coordination." https://wiki.onosproject.org/display/ONOS/Cluster+Coordination (accessed.
[54] "Apache Kudu - Fast Analytics on Fast Data." https://kudu.apache.org/ (accessed April 14, 2019.
[55] "Hashicorp Raft implementation." https://github.com/hashicorp/raft (accessed April 14, 2019.
[56] "The Raft Consensus Algorithm." https://raft.github.io/ (accessed April 14, 2019.
[57] "Barefoot P4 Studio." https://www.barefootnetworks.com/products/brief-p4-studio/ (accessed April 14, 2019.
[58] "P4 v16 Portable Switch Architecture (PSA)." https://p4.org/p4-spec/docs/PSA-v1.0.0.html (accessed April 14, 2019.
[59] "NOPaxos consensus protocol." https://github.com/UWSysLab/NOPaxos (accessed April 14, 2019.

[60] L. Lamport, "Fast paxos," *Distributed Computing,* vol. 19, no. 2, pp. 79-103, 2006.

[61] "Yahoo! Cloud Serving Benchmark in C++, a C++ version of YCSB." https://github.com/basicthinker/YCSB-C (accessed April 14, 2019.

[62] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996, pp. 267-275.

[63] J. D. Valois, "Implementing lock-free queues," in *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*, 1994, pp. 64-69.

[64] S. Prakash, Y. H. Lee, and T. Johnson, "A nonblocking algorithm for shared queues using compare-and-swap," *IEEE Transactions on Computers,* vol. 43, no. 5, pp. 548-559, 1994.

[65] K. Karanasos, S. Rao, C. Curino *et al.*, "Mercury: Hybrid centralized and distributed scheduling in large shared clusters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 485-497.

[66] X. Jin, X. Li, H. Zhang *et al.*, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 121-136.

[67] D. R. Ports and J. Nelson, "When Should The Network Be The Computer?," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019, pp. 209-215.

[68] A. Sapio, I. Abdelaziz, A. Aldilaijan *et al.*, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, 2017, pp. 150-156.

[69] B. Cully, J. Wires, D. Meyer *et al.*, "Strata: High-performance scalable storage on virtualized non-volatile memory," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2014, pp. 17-31.

[70] N. Handigol, M. Flajslik, S. Seetharaman *et al.*, "Aster* x: Load-balancing as a network primitive," in *GENI Engineering Conference (Plenary)*, 2010, pp. 1-2.

[71] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-based server load balancing gone wild," in *Proceedings of the USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Boston, MA, 2011: USENIX Association.

[72] A. K. Nayak, A. Reimers, N. Feamster *et al.*, "Resonance: dynamic access control for enterprise networks," in *Proceedings of the ACM workshop on Research on enterprise networking*, Barcelona, Spain, 2009: ACM, doi: 10.1145/1592681.1592684.

[73] A. J. Mashtizadeh, M. Cai, G. Tarasuk-Levin *et al.*, "XvMotion: unified virtual machine migration over long distance," in *Proceedings of the USENIX Annual Technical Conference*, Philadelphia, PA, 2014: USENIX Association.

[74] A. Lara, A. Kolasani, and B. Ramamurthy, "Network innovation using openflow: A survey," *IEEE communications surveys & tutorials,* vol. 16, no. 1, pp. 493-512, 2014.

[75] X. Jin, X. Li, H. Zhang *et al.*, "Netchain: scale-free sub-RTT coordination," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Renton, WA, USA, 2018: USENIX Association.

[76] H. T. Dang, D. Sciascia, M. Canini *et al.*, "NetPaxos: consensus at network speed," in *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research*, Santa Clara, California, 2015: ACM, doi: 10.1145/2774993.2774999.

[77] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for WANs," presented at the Proceedings of the 8th USENIX conference on Operating systems design and implementation, San Diego, California, 2008.

[78] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in Egalitarian parliaments," presented at the Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, Farminton, Pennsylvania, 2013.

[79] S. J. Park and J. Ousterhout, "Exploiting commutativity for practical fast replication," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 47-64.