Falcon – Low Latency, Network-Accelerated Scheduling

Ibrahim Kettaneh[1], Sreeharsha Udayashankar[1], Ashraf Abdel-hadi[1], Robin Grosman[2], Samer Al-Kiswany[1]

[1]University of Waterloo, Canada [2]Huawei Technologies, Canada

ABSTRACT

We present Falcon, a novel scheduler design for large scale data analytics workloads. To improve the quality of the scheduling decisions, Falcon uses a single central scheduler. To scale the central scheduler to support large clusters, Falcon offloads the scheduling operation to a programmable switch. The core of the Falcon 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 26 times and increase the scheduling throughput by 25 times compared to state-of-the-art centralized and decentralized schedulers.

CCS Concepts

• Networks → Network services → In-network processing;
• Networks → Network services → Cloud computing;

KEYWORDS

Scheduling, programmable networks, large-scale cluster.

1. Introduction

Recent increased adoption of real-time analytics [1, 2] is pushing the limits of traditional data processing frameworks [3]. Applications such as real-time object recognition [4], real-time fraud detection [1], IoT applications [1], and video quality prediction [5] 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 6 million scheduling decisions per second. Furthermore, for such tasks, scheduling delays beyond 1 ms are intolerable.

Traditional data processing frameworks use centralized schedulers [6, 7]. Although the centralized scheduler has accurate knowledge about the utilization of each node in the cluster and can make precise scheduling decisions, it cannot scale to process thousands of status reports from cluster nodes and millions of scheduling decisions [5, 8]. For instance, Firmament [9], the state-of-the-art centralized scheduler can only support a cluster with 100 nodes for short real time tasks [9].

EuroP4 '20, December 1, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-8181-9/20/12…$15.00
https://doi.org/10.1145/3426744.3431322

To overcome the limitations of a centralized scheduler, large scale data analytics engines [8, 10, 11, 12] have adopted a distributed scheduling approach. This approach employs tens of schedulers to increase scheduling throughput and reduce scheduling latency. These schedulers do not have accurate information about the load in the cluster, they either use stale cluster data or probe a random subset of nodes to find nodes to run a given set of tasks [8, 10, 11]. The disadvantage of this approach is that the scheduling decisions are suboptimal, as they are based on partial or stale information, and the additional probing step increases the scheduling delay. Furthermore, this approach is inefficient as it requires using tens of nodes to run the schedulers. For instance, Sparrow uses a scheduler node for every 10 backend nodes and still takes 2 ms to 10 ms to schedule a task [10].

We present Falcon, a scheduling approach that can support large-scale clusters while significantly reducing scheduling latency. Falcon adopts a centralized scheduling approach to eliminate the probing overhead, avoid using tens of scheduling nodes, 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 [13] can forward over 5 billion packets per second, making them ideal candidates for implementing a centralized scheduler for large scale clusters. Unfortunately, leveraging modern switch capability is complicated by their restrictive programming and memory model. In particular, the restrictive memory model allows for performing a single operation on a memory location only 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.

Central to Falcon's design is a novel P4-compatible circular task queue data structure that allows for retrieving a task in one round trip time and supports adding large lists of tasks (§4).

To demonstrate the powerful capabilities of the proposed approach, we built a Falcon prototype. Our evaluation on a cluster with a Barefoot Tofino switch [13] shows that Falcon can reduce scheduling overhead by up to 26 times and for short tasks it improves the task execution time by 14% compared to state-of-the-art scheduler.

2. Overview of Scheduler Design

Modern data analytics frameworks adopt the micro batch scheduling model [6, 9, 10]. The analytics framework submits jobs that consist of m independent tasks (m is typically a small number between 8 and 64). A job is considered complete when all the task in the job have finished execution.

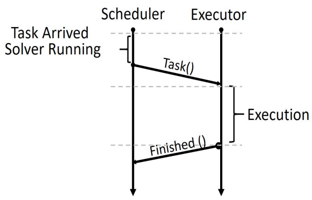

Figure 1. Firmament's scheduling timeline

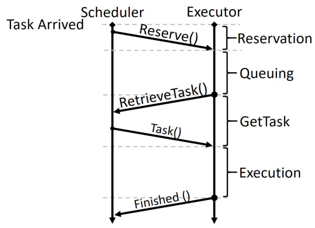

Figure 2. Sparrow's scheduling timeline

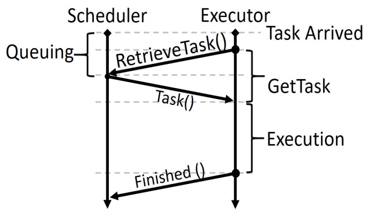

Figure 3. Falcon's scheduling timeline

2.1 Centralized Scheduler Design

Having a single centralized scheduler that maintains accurate cluster status information can result in high quality scheduling decisions [5, 6, 9, 14]. Unfortunately, this design cannot perform millions of scheduling decisions per second to support large scale clusters. For instance, Firmament [9], a state-of-the-art centralized scheduler, models the scheduling problem as a graph with edges extended from tasks to executors that can run them. Firmament uses a min-cost max-flow solver to find the best mapping from tasks to executors. Every time a new job is submitted (Figure 1), the task graph is updated and the graph solver is executed on the new graph. Despite optimizing the solver implementation, the Firmament authors report that it cannot scale beyond a cluster with 1200 CPU cores (100 12-core nodes in their paper) with real time workloads.

Apache Spark [6] also uses a centralized scheduler design. Our evaluation (§5) and the authors of Sparrow [10] show that Spark suffers infinite queuing when task runtime falls below 1.5 seconds.

2.2 Distributed Scheduler Design

Modern distributed schedulers [8, 10, 12] base their scheduling decisions on stale cluster status or on sampling a subset of cluster nodes. For instance, in Sparrow [10], the state of the art distributed scheduler, to schedule a job with *m* tasks, the scheduler submits probes to *2m* randomly selected executors (Figure 2). For instance, if the job has 32 tasks, the scheduler probes 64 out of potentially hundreds of nodes in the cluster. The executors queue the probes. When an executor 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.

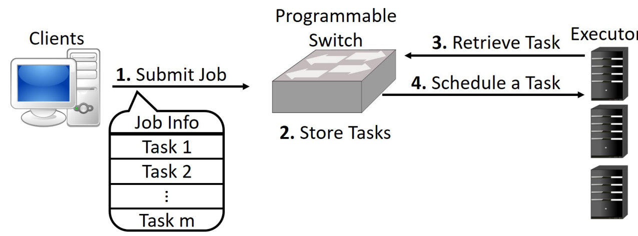

Figure 4. Falcon's architecture

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. Falcon Overview

Falcon is an in-network centralized scheduler that can assign tasks precisely to free executors with minimal overhead. Figure 4 shows Falcon's architecture, which consists of backend nodes, client nodes and a centralized programmable switch.

3.1 Falcon Client

Similar to Spark [6] and Sparrow [10], a data analytics framework groups independent tasks into jobs and submits these jobs to the scheduler. The data analytics framework is a client of the scheduler. In the rest of the paper we use the term client and data analytics framework interchangeably. Once all the tasks in their job finish execution, clients submit their next jobs. As in current data analytics frameworks, clients are responsible for tracking data dependency between tasks and resubmitting failed tasks [6, 10].

3.2 Executors

Figure 3 shows the scheduling steps in Falcon. When an executor becomes free, it sends a message to the scheduler to request a new task. Thus, the scheduler only assigns tasks to free executors, effectively avoiding head-of-line blocking. If the scheduler has no tasks, it sends a no-op task to the executor. The executor waits for a configurable period of time before requesting a task again.

3.3 Programmable Switch

Falcon uses a centralized in-network scheduler. The switch receives job descriptions that include a list of tasks (Figure 4). The switch adds these tasks to a circular queue. The switch assigns a task in first-come-first-serve order to the next executor that requests a task.

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

3.4 Deployment Approach

As with previous projects that leverage switch capabilities [15, 16, 17, 18], the network controller installs forwarding rules to forward all job submission tasks through a single switch; that switch will run the Falcon scheduler. The controller typically selects a common ancestor switch of all

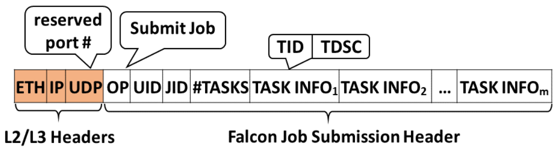

Figure 5. Falcon's job_submission header

executors. While this approach may create a longer path than traditional forwarding, the effect of this change is minimal. Li et al. [17] reported that for 88% of cases, there is no additional latency, and the 99th percentile had less than 5 µs of added latency.

4. System Design

4.1 Network Protocol

Falcon introduces an application-layer protocol embedded in packets' L4 payload. Similar to other systems that use programmable switches [15, 16, 17], 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 an executor. A single job may span multiple job_submission packets. We briefly discuss these headers in this section. The next subsections detail our design.

Figure 5 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.
- #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 any task the system.
- TDESC: the task description that determines the task to be executed.

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

4.2 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.

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 add a task to 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 can only be accessed once per packet, the correction must be made in a future packet. We discuss how to detect and correct incorrect pointers later in this section.

4.3 Handling Job Submission

The client submits a job by populating the header of a job_submission packet (Figure 5) 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 removes the first task from 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 5) 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 again like a new packet. The scheduler removes the first task from the task list (TASK INFO$_1$ in Figure 5) 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 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 packet* to the client. The error packet contains the list of tasks that are not added to the queue. The client then retries submitting a new job after a while.

4.4 Handling Task Retrieval

To avoid head-of-line blocking, executors retrieve tasks only when they become free. To retrieve a task, an executor sends a request to the switch. 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 executor, 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 executor repeats the request after a while.

4.5 Pointer Correction

When the scheduler receives a job submission packet, it executes read_and_increment(add_ptr) first, then checks whether the queue is full. If the queue is full, incrementing 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 retrieval 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. When the next job_submission request is received, the scheduler adds the first task in the queue. The scheduler then checks if the retrieve_ptr needs adjusting, i.e., if the retrieve_ptr is larger than add_ptr. If the retrieve_ptr needs adjusting, the scheduler recirculates a packet and sets the retrieve_ptr to equal the index of the newly added task. A Boolean flag (is_repairing_retrieve_ptr) is set to ensure the scheduler only recirculates one repair packet.

4.6 Fault Tolerance

The switch maintains a soft state. On switch failure a new switch is selected to run the scheduling pipeline. The clients will timeout on all previous submitted tasks and will resubmit those tasks. As with the current frameworks [6, 10] if a tasks fails due to executor or communication failure, the client resubmits the task.

Similarly, if a job submission packet or a task completion packet are lost, the sender will resubmit the packet. This may lead to double execution of a task. As our tasks are idempotent this does not affect correctness but may lead to a loss of efficiency.

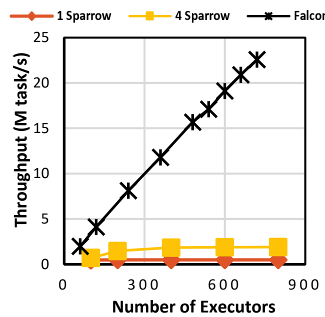

Figure 6. Scheduling Throughput. The y-axis is plotted in millions of tasks per second.

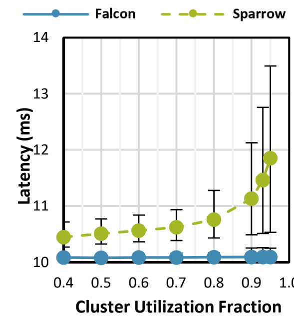

Figure 7. Job Latency for various utilization rates. The error bars depict the 5th and 95th percentiles.

5. Evaluation

We compare the performance of Falcon against that of state of the art centralized and distributed schedulers.

Testbed. We perform all experiments on a 12-node cluster. Each node has 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 use 10 nodes as backend nodes (to host executors) and 2 nodes as client nodes. Unless otherwise specified, each backend node runs 6 executors (i.e., a total of 60 executors). For Sparrow, we run two schedulers on the client nodes, a configuration that is favorable to Sparrow because it reduces communication latency between the client and the collocated scheduler.

Workload. We use a synthetic workload similar to the one used to evaluate Sparrow [10]. Each client submits a job every 10 ms, and each job contains a set of 10-ms tasks. We vary the number of tasks per job to change the system utilization.

Alternatives. We compare Falcon with Sparrow, the state-of-the-art distributed scheduler. Our evaluation of Sparrow reveals that its implementation is not efficient due to using Java and RPCs. We reimplemented Sparrow in C++ using raw sockets. Our C++ implementation achieves up to 25 times higher throughput and 2 times lower latency than the original Java implementation. For the rest of our evaluation we use our C++ implementation of Sparrow.

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 [10]. 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 our figures for clarity.

Finally, we experimented with Firmament. Unfortunately, the Firmament open source implementation could not run our workloads with millisecond tasks. We are currently debugging this deployment. Nevertheless, Firmament authors report that it cannot scale for more than 100 nodes with 5 ms tasks which roughly equates to a peak throughput of under 400k scheduling decisions per second.

5.1 Scheduling Throughput

Figure 6 shows the throughput of Falcon and two configurations of Sparrow C++, with 1 sparrow scheduler and with 4 sparrow schedulers. The throughput of a single sparrow scheduler represents the performance of a highly optimized software-based centralized scheduler. To increase the load on the scheduler we ran no-op tasks and increased the number of executors (shown on x-axis in Figure 6). An executor continuously receives task information, drops it, and then requests a new task. Figure 6 shows that Sparrow, even with 4 schedulers, is not able to support more than 500 executors, with its throughput peaking at 1.9 million scheduling decisions per second. On the other hand, our cluster is too small to stress Falcon enough. With 800 executors, Falcon achieves over 23 million scheduling operations per second (equivalent to over 40 Sparrow schedulers). The switch data sheet indicates that the switch can handle over 5 billion packets per second, indicating that Falcon's performance limit is significantly higher than the workload our no-op executors can generate.

5.2 Job Scheduling Delays

Figure 7 shows the job scheduling delays for various system utilization levels. Falcon significantly reduces the scheduling delay at all utilization levels. At the 95% utilization rate, Falcon reduces the median scheduling delay by 18.5 times (0.09 ms compared to 1.76 ms for Sparrow) and the 95th percentile by 11.8 times (0.25 ms compared to 3.22 ms for Sparrow). Even at 50% utilization, Falcon reduces the scheduling delay by up to 6.8 times. 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 the scheduling overhead increases in Sparrow with larger jobs and higher utilization.

5.4 Scheduling Overhead Breakdown

To understand the performance differences between Falcon and Sparrow, we measure the time spent on each step of both protocols. Figure 8 shows 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 completes all scheduling steps in under 100 μs, whereas Sparrow took up to 3.11 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 26 times higher 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.

6. Related Work

Hybrid Schedulers. A few systems such as Hawk [11] and Mercury [19] use a hybrid scheduling approach. These consist of a centralized scheduler to handle long-running batch jobs and a

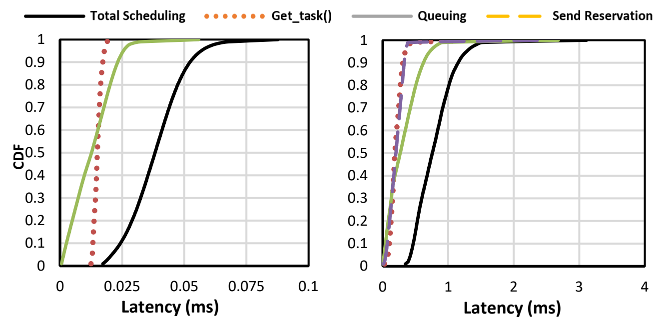

Figure 8. Breakdown of the scheduling delays of Falcon (a) and Sparrow (b). Note the difference in scale of the x-axis between (a) and (b).

distributed scheduler to support low-latency scheduling. Unfortunately, this approach still results in low-quality scheduling decisions.

Network-Accelerated Systems. Recent projects have utilized programmable switches to accelerate consensus protocols [17, 20, 21, 22], implement in-network caching [23], DNN training and inferencing [24], and in-network aggregation operations [25]. R2P2 [26] is the closest of these efforts to our design. R2P2 build a load balancer for RPC calls. R2P2 does not maintain a task queue but rather aims to immediately submit an incoming RPC to a server. If no server is available, R2P2 recirculates the packet until a server becomes available. This approach is not efficient with data analytics workloads that experiences burstiness in task arrivals. Furthermore, R2P2 does not guarantee FIFO ordering or scheduling decision optimality. Falcon presents a P4-compatible queue design that overcomes these inefficiencies.

7. Concluding Remarks and Future Work

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

In our ongoing work, we are extending the scheduler to support three common scheduling policies: data locality-aware scheduling, priorities, and scheduling on nodes with specific resources. Furthermore, we are building a simulator to evaluate Falcon at large scale.

Acknowledgement

We thank the anonymous reviewers and Khuzaima Daudjee for their insightful feedback. This research was supported by an NSERC Collaborative Research and Development grant and Waterloo-Huawei Joint Innovation lab grant.

8. References

[1] Stonebraker, M., Çetintemel, U. and Zdonik, S. The 8 requirements of real-time stream processing. ACM Sigmod Record, 34, 4 (2005), 42-47.

[2] Wang, S., Liagouris, J., Nishihara, R., Moritz, P., Misra, U., Tumanov, A. and Stoica, I. Lineage stash: fault tolerance off the critical path, 2019.

[3] Ousterhout, K., Panda, A., Rosen, J., Venkataraman, S., Xin, R., Ratnasamy, S., Shenker, S. and Stoica, I. The case for tiny tasks in compute clusters, 2013.

[4] Zhang, T., Chowdhery, A., Bahl, P., Jamieson, K. and Banerjee, S. The design and implementation of a wireless video surveillance system, 2015.

[5] Venkataraman, S., Panda, A., Ousterhout, K., Armbrust, M., Ghodsi, A., Franklin, M. J., Recht, B. and Stoica, I. Drizzle: Fast and adaptable stream processing at scale, 2017.

[6] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I. Spark: Cluster computing with working sets. HotCloud, 10, 10-10 (2010), 95.

[7] Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters, 2004.

[8] Ren, X., Ananthanarayanan, G., Wierman, A. and Yu, M. Hopper: Decentralized speculation-aware cluster scheduling at scale, 2015.

[9] Gog, I., Schwarzkopf, M., Gleave, A., Watson, R. N. M. and Hand, S. Firmament: Fast, Centralized Cluster Scheduling at Scale. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016)

[10] Ousterhout, K., Wendell, P., Zaharia, M. and Stoica, I. Sparrow: distributed, low latency scheduling, 2013.

[11] Delgado, P., Dinu, F., Kermarrec, A.-M. and Zwaenepoel, W. Hawk: Hybrid datacenter scheduling, 2015.

[12] Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M. and Zhou, L. Apollo: Scalable and coordinated scheduling for cloud-scale computing, 2014.

[13] Tofino-2 Second-generation of World's fastest P4-programmable Ethernet switch ASICs.

[14] Garefalakis, P., Karanasos, K. and Pietzuch, P. Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications. In Proceedings of the SoCC '19: Proceedings of the ACM Symposium on Cloud Computing, 2019)

[15] Al-Kiswany, S., Yang, S., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H. NICE: Network-integrated cluster-efficient storage, 2017.

[16] Li, X., Sethi, R., Kaminsky, M., Andersen, D. G. and Freedman, M. J. Be fast, cheap and in control with SwitchKV, 2016.

[17] Li, J., Michael, E., Sharma, N. K., Szekeres, A. and Ports, D. R. Just say NO to paxos overhead: Replacing consensus with network ordering, 2016.

[18] Ports, D. R. K., Li, J., Liu, V., Sharma, N. K. and Krishnamurthy, A. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), 2015)

[19] Karanasos, K., Rao, S., Curino, C., Douglas, C., Chaliparambil, K., Fumarola, G. M., Heddaya, S., Ramakrishnan, R. and Sakalanaga, S. Mercury: Hybrid centralized and distributed scheduling in large shared clusters, 2015.

[20] Takruri, H., Kettaneh, I., Alquraan, A. and Al-Kiswany, S. FLAIR: Accelerating Reads with Consistency-Aware Network Routing, 2020.

[21] Jin, X., Li, X., Zhang, H., Foster, N., Lee, J., Soulé, R., Kim, C. and Stoica, I. Netchain: Scale-free sub-rtt coordination, 2018.

[22] Dang, H. T., Sciascia, D., Canini, M., Pedone, F. and Soulé, R. Netpaxos: Consensus at network speed, 2015.

[23] Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C. and Stoica, I. Netcache: Balancing key-value stores with fast in-network caching, 2017.

[24] Ports, D. R. and Nelson, J. When Should The Network Be The Computer?, 2019.

[25] Sapio, A., Abdelaziz, I., Aldilaijan, A., Canini, M. and Kalnis, P. In-network computation is a dumb idea whose time has come, 2017.

[26] Kogias, M., Prekas, G., Ghosn, A., Fietz, J. and Bugnion, E. R2P2: Making RPCs first-class datacenter citizens. In Proceedings of the 2019 USENIX Annual Technical Conference (ATC 19), 2019)