In-Network Scheduling for Real-Time Analytics

by

Sreeharsha Udayashankar

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, 2021

AUTHOR'S DECLARATION

This thesis contains materials I have authored or co-authored. Please refer to the Statement of Contributions for details. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

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

Statement of Contributions

The results of this work are included in two papers. The first, is a position paper at the third P4 Workshop in Europe (EuroP4 ’20) [1] and the second is a main paper under submission to the Symposium on Cloud Computing 2021. I am the first author of the main paper and it has been done in collaboration with Ibrahim Kettaneh and Ashraf Abdel-Hadi under the supervision of our advisor, Prof. Samer Al-Kiswany. Ibrahim was involved in the basic FIFO Bolt design (Section 3.2). Ashraf was involved in the FIFO Bolt design as well as the evaluation of our Bolt prototype (Chapter 5).

Abstract

This thesis presents Bolt, a novel scheduler design for large-scale real-time data analytics. Bolt achieves the scheduling accuracy of modern centralized schedulers while supporting clusters with hundreds of thousands of nodes. At Bolt's core is a scheduler design that leverages modern programmable switches. Bolt supports a FIFO scheduling policy, as well as task priority-based and task resource constraint-based scheduling policies.

Evaluation of a Bolt prototype on our cluster with a Barefoot Tofino switch shows that the proposed approach can reduce scheduling overhead by 40x and increase the scheduling throughput by 50x compared to state-of-the-art centralized and decentralized schedulers.

Acknowledgements

I would like to take this opportunity to express my appreciation towards my advisor, Prof. Samer Al-Kiswany for his great mentorship. Without his continuous support and guidance, this thesis would not have been a possibility.

I would like to thank my collaborators and friends, Ibrahim Kettaneh and Ashraf Abdel-Hadi, without whom this thesis would not be complete. I would also like to thank everybody at the Waterloo Advanced Systems Lab for the positive environment they have fostered.

Finally, I would like to thank my parents, family and friends for their unconditional motivation and support. There are no words which can express my gratitude to them.

Table of Contents

List of Figures

Chapter 1– Introduction

Online Data-Intensive (OLDI) services [2, 3, 4] are one of the fundamental building blocks of today's internet infrastructure. These are workloads which perform computations over massive datasets but still aim to provide fast response times in the order of milliseconds. Numerous applications such as low latency web-services [5], real-time analytics [6, 7, 8], algorithmic smart trading [7, 9, 10] and rapid object detection [11] fall under this banner as they require real-time responses ranging from tens to hundreds of milliseconds. The service times for tasks comprising these workloads are proportionally tiny [12] and we expect these to go even lower given the benefits. For instance, it is reported that lowering trade latency by a millisecond can boost a firm's earnings by upwards of a hundred million dollars each year [13].

The traditional bottlenecks for sub-second tasks within data processing frameworks have been CPU and I/O related [14] and various systems such as MemCache [15] and RAMCloud [16] have already attempted to address these. However, this no longer holds true when we move an order of magnitude lower into the millisecond range. Chen et al. [17] have recently demonstrated that task scheduling delays are the major component of end-to-end response times for low-latency analytics. In their research, they have observed that scheduling delay can account for nearly 60% of total job execution time when jobs primarily consist of low-latency tasks.

Schedulers for clusters running tasks which take tens of milliseconds must be able to perform a huge number of scheduling decisions per second. These decisions must also be made with extremely low overheads, as scheduling delays beyond 1 ms are intolerable when handling these tasks.

Centralized Schedulers. Traditional data-processing frameworks use centralized scheduler designs [18, 19, 20]. Although such a centralized scheduler can make accurate scheduling decisions with accurate cluster information, they cannot scale to handle large cluster sizes. For instance, Firmament [18], a state-of-the-art centralized scheduler can only support of a cluster of 100 nodes when running real-time tasks and the centralized Spark [19] scheduler cannot support any sub-second tasks.

Distributed Schedulers. To overcome the scalability limitations associated with centralized schedulers, distributed scheduling designs [21, 22, 23] were adopted by many frameworks. This approach utilizes multiple schedulers operating autonomously on a cluster with virtually zero intercommunication. As a result, they either operate on stale cluster information [21] or rely on probing a subset of nodes in the cluster to launch their tasks [22, 23]. These scheduling decisions are suboptimal due to their reliance on approximate cluster information, and probing can add onto the scheduling overhead, violating the tight bounds needed for real-time tasks. Furthermore, a large percentage of additional nodes need to be dedicated to running these schedulers. For instance, a single Sparrow [23] scheduler can handle only a few tens of nodes running real-time tasks while also possessing a scheduling overhead of 2-10 ms, thus requiring a large number of machines reserved for running it while servicing even medium-sized clusters of hundreds of nodes.

Bolt. This thesis presents Bolt, a scheduler that can make rapid real-time scheduling decisions based on global cluster information. To make these scheduling decisions precise, Bolt follows a centralized scheduling paradigm. It can also scale to support hundreds of thousands of nodes when running real-time tasks as it possesses an extremely high scheduling throughput. Bolt possesses the ability to prioritize critical tasks as well as handle tasks requiring specific resources such as GPUs in a fashion similar to the Hadoop MapReduce [20] and Spark [19] schedulers. Additionally, Bolt provides these features at a finer task granularity vis a vis the job granularity in the aforementioned schedulers.

To improve the scheduling performance and support large clusters, Bolt accelerates scheduling decisions by leveraging modern programmable switches [24, 25, 26]. Modern programmable switches can forward more than 4 billion packets per second, making them ideal candidates for implementing a centralized scheduler for large scale clusters. However, leveraging their capabilities is challenging due to their restrictive programming and memory models. 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 because standard queue operations access the queue size variable twice: once to check whether the queue is empty or full, and once to increment or decrement its size.

The key to Bolt's design is a novel P4-compatible circular task queue that enables task retrieval at line-rate while also supporting the addition of large lists of tasks (Section 3.2). Furthermore, despite the limitations of current switches, Bolt's design supports common scheduling policies, including priority and resource constraints (i.e., schedule a task on nodes with certain resources).

The rest of this thesis is organized as follows. Chapter 2 provides the motivation and relevant background for this thesis. Chapter 3 provides an overview of the Bolt end-to-end system design. In Section 3.2, the design details needed to support a FIFO scheduling policy with an in-network scheduling paradigm are discussed. Section 3.3 and Section 3.4 then go ahead and discuss how this design can be extended to support task priority-based and task resource constraint-based scheduling policies. Chapter 4 describes our Bolt prototype implementation as well as provides an outline of the other state-of-the-art schedulers we have compared Bolt against. In Chapter 5, we compare our Bolt prototype against other schedulers to demonstrate its scheduling latency and throughput improvements. Lastly, in Chapter 6, we present other work related to Bolt, and outline how Bolt differs from them.

Chapter 2 – Background

This section aims to provide the background information required for the topics discussed in this thesis.

2.1 Real-Time Workloads

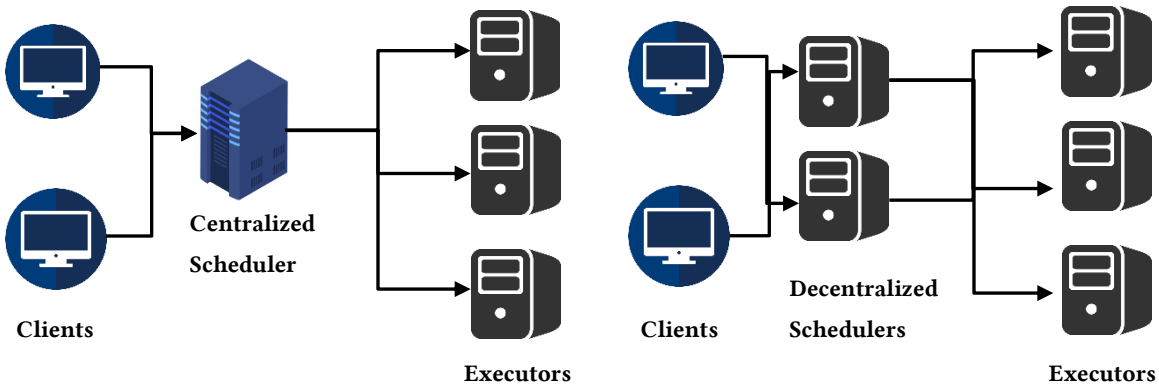
Real-Time Workloads are those that require end-to-end response times ranging from the tens to hundreds of milliseconds. Online Data Intensive (OLDI) workloads are workloads which meet this constraint while performing parallel computations on massive data sets. Web services which require response times in the sub 100ms and sub 10 ms range have been studied extensively before [2, 3, 4, 5] and are widely used as the representatives for this class of workloads. For instance, the Google Web-Search system [27] updates query results interactively, displaying search results within a few tens of milliseconds of a user typing search queries . Dean et al [5] also envision that upcoming augmented-reality applications will require web services with even lower response times.

Real-time analytics are a fast-growing area of interest and are being applied in various areas such as traffic analysis [6], financial analytics [7] and smart-grid monitoring [28, 29]. With growing trends towards leveraging the Internet-of-Things (IoT) and analytics in areas such as defense [30] and agriculture [31], we expect an explosive growth of real-time analytics in the future. Financial analytics in particular benefits the most from latency improvements, as the lowering of trade delays by even a millisecond can boost firm earnings by upwards of a hundred million dollars a year [13] .

Platforms for emerging AI applications [32, 33] powered by techniques such as reinforcement learning must support fine-grained computations in the order of milliseconds (such as rendering actions when interacting with the real world). As noted by the authors of Ray [32], such frameworks handle millions of tasks per second but require millisecond-level latency constraints.

User facing services demand tight latency bounds, which can severely impact user experience when breached. A previous study on the impact of response times [34] has shown that anything over an end-to-end response time of 150 ms degrades the quality of such services noticeably. Finally, real-time applications such as rapid object detection [11, 35] and augmented reality applications [5] consist of short latency dependent tasks and would benefit immensely from response-time improvements.

To run these applications even on moderate sized clusters with a few hundreds of nodes, the data analytics framework needs to make hundreds of thousands of precise scheduling decisions per second while keeping scheduling latencies below a single millisecond.



(a) Centralized Scheduler Design (b) Decentralized Scheduler Design

Figure 1. Scheduling Paradigms

2.2 Scheduling Paradigms

Modern data-processing frameworks adopt the micro-batch scheduling model [19, 20]. Jobs are submitted that consist of *m* independent tasks (*m* is typically a small number between 8 and 128). Tasks are run in parallel by executors running on cluster nodes. A job is considered complete only when all the tasks within have completed their execution.

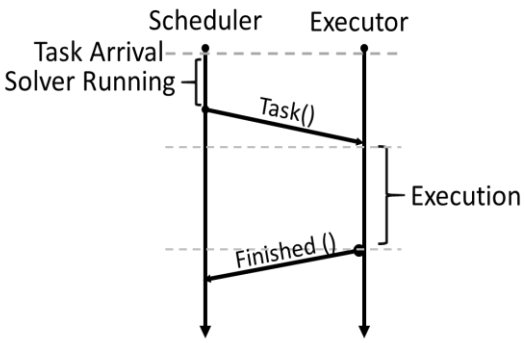


Figure 2. Firmament's Scheduling Timeline

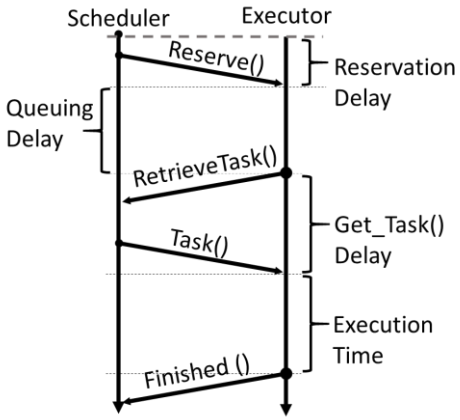


Figure 3. Sparrow's Scheduling Timeline

2.2.1 Centralized Scheduler Design

Centralized scheduler architecture is illustrated in Figure 1a. Having a single centralized scheduler that maintains accurate cluster status information can result in high-quality scheduling decisions. However, this approach cannot support real-time analytics even on moderately sized clusters. For instance, Firmament [18], 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. Each time a new job is submitted, the task graph is updated, and the graph solver is executed on the new graph (Figure 2). Despite optimizing their solver implementation, the Firmament authors report that it cannot scale beyond a cluster with 1200 CPU cores (100 12-core nodes in their paper) when running real-time workloads.

Apache Spark [19] also uses a centralized scheduler design. Our evaluation (Chapter 5) and the authors of Sparrow [23] show that the Spark scheduler breaks down and suffers infinite queuing when task execution time falls below 1.5 seconds.

2.2.2 Decentralized Scheduler Design

Modern distributed schedulers [21, 22, 23] aim to support large-scale clusters by employing tens of schedulers. Decentralized scheduler architecture is illustrated in Figure 1b. To avoid the high overhead of coordinating the scheduling decisions between multiple schedulers,

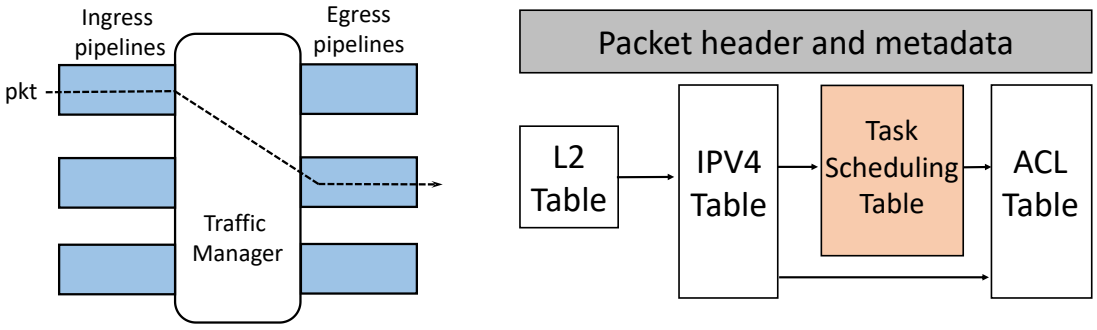
distributed schedulers base their scheduling decisions on partial or stale cluster status information. For instance, in Sparrow [23], a state-of-the-art distributed scheduler, to schedule a job with *m* tasks, the scheduler submits probes to *2m* randomly selected executors (Figure 3). If the job has 32 tasks, the scheduler probes just 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, because the scheduler does not have complete knowledge of the cluster utilization. Hopper [22] adopts a similar approach. This approach leads to suboptimal scheduling decisions because the scheduler only probes a fraction of the cluster nodes, and the probing step increases the scheduling delay.

Apollo [21] is a distributed scheduler that uses a central resource monitoring service to monitor the nodes. Each scheduler queries the resource-monitoring service to identify vacant nodes. The resource-monitoring service loosely coordinates between schedulers. Consequently, it is likely that multiple schedulers will schedule tasks on the same set of nodes, leading to less efficient scheduling decisions.

2.2.3 Programmable Switches

Programmable switches facilitate the implementation of an application-specific packet-processing pipeline that is deployed on network devices and executed at line speed. A number of vendors produce network-programmable ASICs, including Barefoot's Tofino [24, 25] and Broadcom's Trident 3 [26].

Figure 4.a illustrates the basic data plane architecture of modern programmable switches. The data plane consists of 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, which finally emits the packet.



(a) Switch data plane

(b) Pipeline for implementing in-network scheduling

Figure 4. Programmable Switch Architecture

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, then the corresponding action is executed. Each stage has dedicated resources, including tables and register arrays (a memory buffer). Stages can share data through the packet headers 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. Packet processing can be viewed as a graph of match-action stages.

Programmers use domain-specific languages such as P4 [36, 37] to define their packet headers, define tables, implement custom actions, and configure the processing pipeline.

Challenges. The need to execute custom actions at line speed restricts what modern ASICs can do. 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 or written per packet per register, and (5) the size of the per-packet metadata passed between stages. In addition, modern ASICs lack support for loops or recursion.

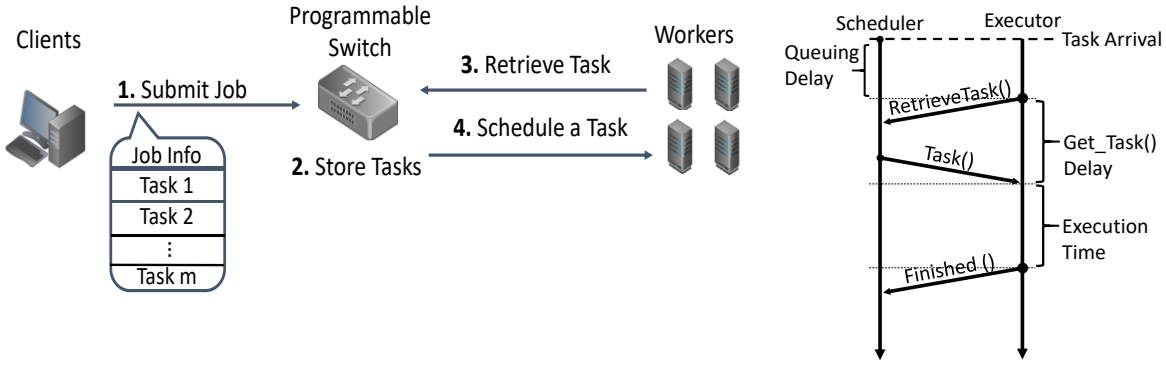
The restrictive memory model constitutes a particular challenge to building an in-network scheduler. A memory register (the only memory that can preserve variables across packets) can only be accessed in a single stage and using a single operation. The operation can be either

a simple read or write or perform a single arithmetic operation (e.g., read and increment or read and set).

Chapter 3 – Bolt's Design

3.1 Design Overview

Bolt is an in-network centralized scheduler that precisely assigns tasks to free executors with minimal overhead. An executor is a process running on a worker node and receives tasks from the scheduler to execute. Multiple executors run on a single worker node. Typically, a worker node runs a number of executors equal to the number of the available logical cores (i.e., hardware threads). Figure 5.a shows Bolt's architecture, which consists of clients, executors, and a centralized programmable switch.



(a) Design Architecture (b) Scheduling Timeline

Figure 5. Bolt's Design

3.1.1 Bolt Client

Similar to Spark [19] and Sparrow [23], a data-processing framework groups independent tasks into jobs and submits these jobs to the scheduler. The data-processing framework is a client of the scheduler. In the rest of the paper, we use the terms *client* and *data-processing framework* interchangeably. As in current data-processing frameworks, jobs consist of independent tasks that can run in parallel, and clients are responsible for tracking data dependency between tasks of different jobs [19, 20, 23]. If a task fails, then clients resubmit failed tasks [19, 23].

3.1.2 Executors

Figure 5.b shows the scheduling steps in Bolt. 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. This is essential to meet the latency requirements of low-latency tasks. The executor then completes its task and sends a completion response back to the client via the scheduler.

If the scheduler has no tasks, then it sends a no-op task to the executor. The executor waits for a configurable period of time before requesting a task again.

3.1.3 Programmable Switch

Bolt hosts the centralized in-network scheduler on a programmable switch [24, 25]. The scheduler receives job descriptions that include a list of tasks (Figure 5.a). These tasks reside in memory until an executor is available to run them. The scheduler adds these tasks to a circular queue along with the information needed to identify the client that submitted them. When an executor asks for a task, the scheduler assigns a task to it based on various parameters such as task priority and resource requirements.

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

3.1.4 Deployment Approach

Similar to previous projects that leverage switch capabilities [38, 39, 40, 41], the network controller installs forwarding rules to forward all job-submission requests through a single switch; that switch will run the Bolt scheduler. The controller typically selects a common ancestor switch of all worker nodes. While this approach may create a longer path than traditional forwarding does, the effect of this change is minimal. Li et al. [39] report that for 88% of cases, this approach does not increase the request latency, and the 99th percentile had fewer than 5 μs of added latency.

3.1.5 Fault Tolerance

Bolt follows the failure semantics of modern data analytics engines such as Spark [19]. The Spark application driver handles executor monitoring and, upon detecting executor failure, requests new executors from the cluster manager such as YARN [42] or Mesos [43]. Timeouts and resubmissions are used to handle other issues such as messages lost in transit between different components of the system.

The scheduler maintains a soft state. On switch failure, a new switch is selected to run the scheduling pipeline. The clients will time out on all previous submitted tasks and will resubmit those tasks. Similar to other current frameworks [19, 20, 23], if a task fails due to executor or communication failure, the client times out for this task and resubmits the task.

Similarly, if a job-submission packet or a task-completion packet is lost, the sender will resubmit the packet. This may lead to the double execution of tasks. Because tasks are idempotent [20], this does not affect correctness but may lead to a small loss of efficiency.

3.2 A FIFO Scheduling Design for Bolt

We first present the base design for Bolt's scheduler with a FIFO scheduling policy, then extend this design with priority (Section 3.3) and resource constraint scheduling (Section 3.4).

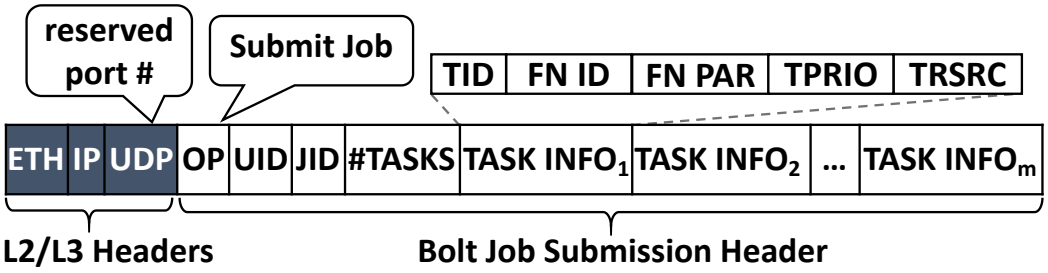


Figure 6. Bolt's job submission packet structure

3.2.1 Network Protocol

Bolt introduces an application-layer protocol embedded in packets' L4 payload. Similar to other systems that use programmable switches [38, 39, 40] , Bolt uses UDP to reduce operation latency and simplify the scheduler design.

Bolt introduces two new packet headers: job_submission, which is used to submit a new job to the scheduler, and a 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 6 shows the main fields of the job_submission packet:

- OP: The request type indicating this is a job submission.
- 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 scheduler 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 in the system.
- TDESC: The task description that determines the task to be executed and its parameters.
- TPRIO: The priority of the task.
- TRSRC: The resource constraints for the task

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

3.2.2 Scheduler Design

Bolt 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 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 index part points to an entry in the queue. The round_num counts the number of rounds the pointer traversed the entire queue. This round_num field helps to resolve special cases when the queue is full or empty.

To detect whether the queue is full or empty, we subtract retrieve_ptr from add_ptr. If the difference is zero, then the queue is empty. If the difference is equal to or larger than the queue size, then the queue is full. In some cases, the difference is negative, meaning retrieve_ptr is larger than add_ptr, in which case the pointers need an adjustment. We discuss this below.

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

Because it can access a pointer only once per packet, Bolt uses an atomic read_and_increment(add_ptr) operation to read add_ptr and increment it in one access. It then checks whether the queue is full. If the queue is not full, then Bolt 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, Bolt 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 (§3.2.5).

3.2.3 Handling Job Submissions

The client submits a job by populating the header of a job_submission packet (Figure 6) and sending the packet to the scheduler. The scheduler 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, Bolt checks the #TASKS field in the packet. If it is larger than zero, then 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 6) contains a list of tasks. To add multiple tasks to the queue, Bolt 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. Bolt 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, then 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.

3.2.4 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 scheduler. The scheduler calls read_and_increment(retrieve_ptr) and reads one task from the queue. If the task's is_valid flag is True, then 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; then this indicates that the queue is empty. In this case, a no-op task is sent back to the executor, which

repeats the request after waiting a configurable length of time. We are exploring a mechanism to keep track of vacant nodes in order to reduce the number of task retrieval requests.

3.2.5 Pointer Correction

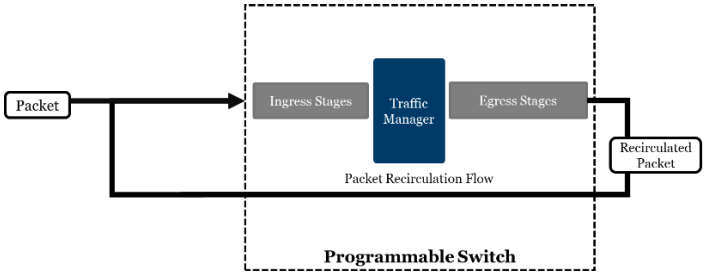


Figure 7. Packet Recirculation

When the scheduler receives a job submission packet, it executes read_and_increment(add_ptr) first, then checks whether the queue is full. If the queue is full, incrementing add_ptr was a mistake. To correct this mistake, the scheduler recirculates a repair packet to reset add_ptr to its original value. To avoid a case in which multiple job_submission packets try to reset add_ptr, we add 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), then 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 whether retrieve_ptr needs adjusting (i.e., whether retrieve_ptr is larger than add_ptr). If retrieve_ptr needs adjusting, the scheduler recirculates a packet and sets 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.

It is important to note that incrementing the retrieve_ptr will never cause queue overflows as each pointer is a combination of a round number and an index, as noted in §3.2.2. The round number is used to prevent queue overflows from occurring.

3.3 Priority Based Scheduling

Priority-based scheduling is a common scheduling approach in modern schedulers [18, 19, 21, 23]. For higher flexibility, unlike current frameworks (Hadoop [20] and Spark [19]) that support priority-based scheduling at the job level, Bolt offers priority-based scheduling at the task level, meaning tasks within the same job may have different priorities. Tasks within the same priority level are executed in FIFO order.

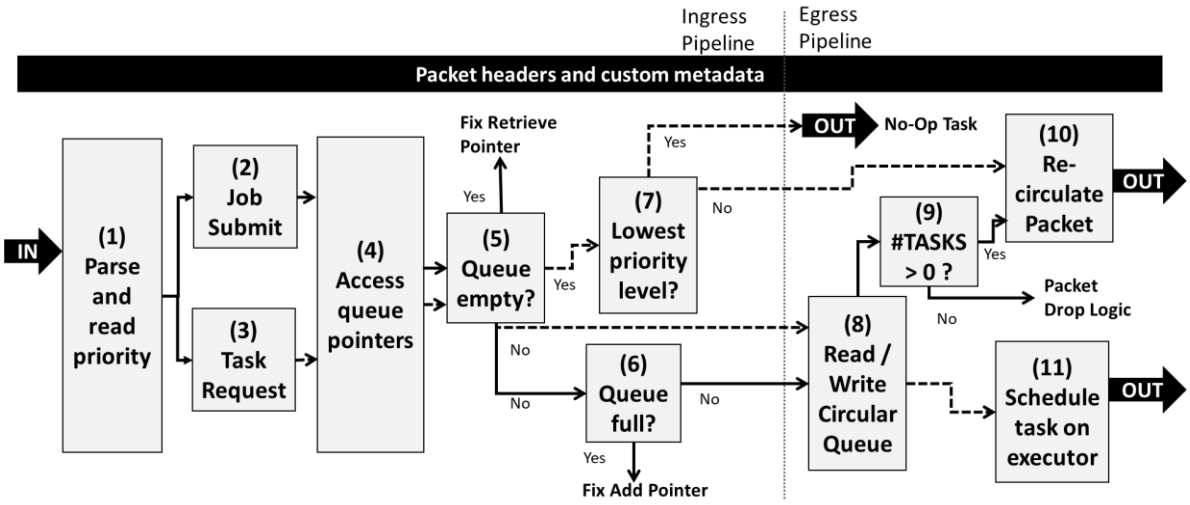


Figure 8. Logical view of the Bolt switch data plane for priority-based scheduling.

Note that the dashed lines represent task_retrieval flow and solid lines represent job_submission flow.

Figure 8 shows the logical view of Bolt's data plane with priority scheduling. To support multiple priority levels, we use a separate FIFO queue (described in the previous section) for each priority level. Higher priority levels have lower priority numbers, with priority 1 being the highest priority.

3.3.1 Job Submission

A client indicates a task's priority level in the TPRIO field in the TASK_INFO field of the job submission packet (Figure 6). When the scheduler receives a job_submission packet, it forwards it to the queue that matches the priority level in the TPRIO field (Step 4 in Figure 8). The scheduler adds the task to the queue as detailed in §3.2.3.

3.3.2 Task Retrieval

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

The scheduler will retrieve a task from the FIFO queue corresponding to the priority level specified in the RTRV_PRIO field (Step 8 in Figure 8). If the retrieved task is a valid task, the scheduler forwards the task to the executor. If the retrieved task is an invalid task, indicating that the selected queue is empty, the scheduler will increment the RTRV_PRIO field and recirculate the packet (Steps 7 and 10 in Figure 8). Incrementing RTRV_PRIO makes the scheduler retrieve a task from a lower priority queue. If RTRV_PRIO becomes larger than the number of priority levels in the system, indicating that there are no tasks at any priority level, the scheduler sends a no-op packet to the executor. The executor retries after a configurable time period. Adjusting retrieve_ptr and add_ptr follows the same logic presented in Section 3.2.5.

Recirculation Overheads. In the worst case, Bolt may recirculate a retrieve_task packet up to the number of priority levels supported in the system. In Chapter 5, we show that this adds negligible overhead, because a single packet recirculation takes less than a microsecond.

3.4 Scheduling with Task Resource Constraints

Priority Level 1	
Resources Required	**Task**
R1 & R3	Task 2
R2	Task 3
R3	Task 4
...	...

Priority Level 2	
Resources Required	**Task**
R3	Task 7
R2 & R3	Task 9
R3	Task 15
...	...

Priority Level 3	
Resources Required	**Task**
R1	Task 21
R2	Task 29
R2 & R3	Task 31
...	...

Figure 9. Bolt queue design with resource constraint and task priority support

Tasks may require specific resources, such as a GPU or large memory. In this section, we extend Bolt's design to handle resource constraints. Similar to MapReduce [20] and Spark [19], Bolt supports binary task constraints, i.e., the task either needs a resource or not. Tasks may have multiple constraints. To simplify the discussion, we present a design with a single priority level.

In our current implementation, Bolt supports eight different kinds of resources. Each submitted task has a TRSRC field that specifies the resources the task needs. TRSRC is a byte representing an 8-bit resource bitmap.

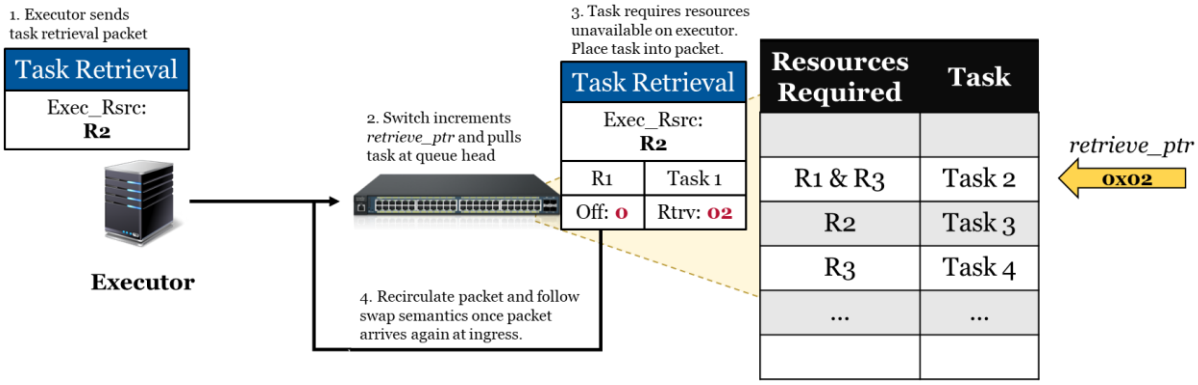


Figure 10. Usage of packet recirculation for resource-constraint scheduling

3.4.1 Job Submission

A client sets the appropriate flags in a task's TRSRC field to specify the required resources. The scheduler uses the same logic described in Section 3.2.3 to process the job_submission packet.

3.4.2 Task Retrieval

The scheduler aims to assign a task to the first executor that has the requested resources. When an executor sends a retrieve_task packet (Figure 10. Step 1) it specifies the resources that it has in an 8-bit bitmap called EXEC_RSRC.

When the scheduler receives a retrieve_task packet from an executor, it retrieves a task from the queue following the process described in Section 3.2.4. The retrieve process increments retrieve_ptr and fetches a task from the queue (Figure 10. Step 2). If the queue is empty, a no-

op task is sent to the executor. If a valid task is retrieved, the scheduler will compare the task's TRSRC field to the executor's EXEC_RSRC field. If the executor has the resources required to run the task, the scheduler sends the task to the executor.

If the retrieved task cannot be executed by this executor, we need to reinsert the task into the task queue and retrieve another one. We achieve this by swapping the retrieved task with the next task in the queue. To do so, the scheduler creates a special swap_task packet (Figure 10. Step 3). The swap_task packet has the following fields: TASK_INFO of the retrieved task; SWAP_IDX, the index of the next entry in the queue; the EXEC_RSRC; the executor's IP address; and the current value of retrieve_ptr stored in pkt_retrieve_ptr. The scheduler then populates and recirculates the swap_task packet (Figure 10. Step 4).

3.4.3 Swapping a Task

When the scheduler receives a swap_task packet, it typically swaps TASK_INFO in the packet with TASK_INFO in the queue at the index specified in the SWAP_IDX field. The swap_task packet does not increment retrieve_ptr. If the swapped task can run using the resources specified in EXEC_RSRC, the task is sent to the executor. Otherwise, the scheduler repeats the swap logic by incrementing SWAP_IDX and then populates and recirculates the swap_task packet.

The scheduler keeps recirculating a swap packet until it either finds a task that can run on this executor or it reaches the end of the queue. If the SWAP_IDX in the packet is larger than add_ptr, indicating that no task in the queue can run on this executor, the scheduler treats the swap_task packet as a job_submission packet, inserts the task following the logic in Section 3.2.3, and sends a no-op task to the executor.

To avoid complex concurrency scenarios in which the scheduler receives multiple retrieve_task packets, the swap_task packet contains the current retrieve_ptr. If the scheduler receives a swap_task packet with a value of a pkt_retrieve_ptr that does not match the current retrieve_ptr, then the scheduler will ignore the SWAP_IDX value and will swap the original task with the task at the head of the queue. This effectively resets the search to the beginning of the queue.

Chapter 4 – Implementation

We have implemented a Bolt prototype using C++. We have implemented the Bolt scheduling logic on a Barefoot Tofino [25] switch using the P4 [36, 37] programming language in 1500 lines of code. The prototype uses executors developed in C++ and follows the job submission model adopted by Sparrow [23]. A C++ client acts as a stub for modern data-analytics frameworks, submitting jobs with configurable sizes, task durations, and interarrival times.

The switch available to us is one of the earliest models of P4 programmable switches and has limited resources. Recent Barefoot Tofino switches have a significantly larger memory and number of stages [24]. Due to the limitations of our switch, our prototype has a queue size of 128K tasks. Hence, we use a 17-bit index and 15-bit round_num within each queue pointer (add_ptr and retrieve_ptr). The prototype can currently support four priority levels. Our back of the envelope calculations show that the newer switches can support one-million-entry queue sizes with larger number of priorities.

The majority of the control logic is currently located in the Ingress pipeline on the switch while the Egress pipeline is used to hold the registers representing the actual task queues themselves. However, there is one exception to this. To handle task constraints, the TRSRC fields are stored in the Ingress pipeline. This is done because if we want to recirculate a packet, it needs to be forwarded to a specific output port, which can only be performed in the Ingress pipeline.

The switch possesses a Linux-based controller which can be used to insert entries into any of the Match-Action tables. We use this controller to plug in the rules to manage packet flow through the switch when Bolt is deployed on it. More importantly, the table entries for Layer 3 and Layer 4 packet switching are also deployed during this phase by the controller. Both of these are one-time setup operations which need to be repeated only upon Bolt re-deployment.

We have performed a few optimizations in our prototype implementation. As noted in Section 3.2.5, two flags are needed to indicate that the add_ptr and retrieve_ptr are being fixed by a recirculating packet. In our prototype, we would need two such flags per priority level,

resulting in a total of 8 flags for the 4 priority levels provided. These flags have been combined into an 8-bit bitmask, with one bit for each flag, to save pipeline stages.

Chapter 5– Evaluation

We compare the performance of Bolt against that of state-of-the art centralized and decentralized schedulers using a combination of synthetic and real-world workloads. Our evaluation aims to address the following questions:

a) How does Bolt's scheduling overhead compare to state-of-the-art centralized and decentralized schedulers?
b) Can Bolt scale to handle large cluster sizes while maintaining the stringent latency requirements required by real-time tasks?
c) What are the benefits achieved by moving the scheduling decisions to programmable switches viz a viz using a software-based scheduler?

5.1 Evaluation Setup

5.1.1 Testbed

We perform all our experiments on a 13-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 [25]. We use 11 nodes as worker nodes and up to two nodes as schedulers (if required) in our experiments.

5.1.2 Alternative Schedulers

We compare the throughput and latency of the following scheduling approaches.

- **Sparrow.** Sparrow [23] is a state-of-the-art distributed scheduler that uses probing to find a vacant node for the next task. Our evaluation of the open-source Sparrow implementation [44] shows that its implementation is not efficient due to using Java and RPCs. We re-implemented Sparrow in C++ using raw UDP 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.
- **Bolt.** We use our Bolt prototype implementation.

- **Bolt-Server.** We implemented a highly tuned centralized scheduler following the Bolt scheduling protocol using C++. This alternative represents an optimized and low overhead implementation of a centralized scheduling approach.

We have evaluated Spark's [19] scheduling delay in our experiments. Unfortunately, Spark could not handle sub-second tasks; this confirms a similar observation made by the authors of Sparrow [23]. The scheduling delay at 50% cluster utilization was 3 seconds. Above 50% utilization, the scheduler could not keep up and experienced infinite queueing. Hence, we did not include Spark in our figures for clarity. Additionally, we experimented with Firmament [18]. Unfortunately, the Firmament open-source implementation could not run workloads consisting of tasks with execution times less than one second. Nevertheless, Gog et al. [18] report that Firmament cannot scale to more than 100 nodes with 6 physical cores each (1200 executors total), when running 5-ms tasks. This roughly equates to a peak throughput of 240K scheduling decisions per second.

5.1.3 Workload

We evaluate our Bolt prototype with a combination of workloads, consisting of the Google Cluster Traces [45] as well as two synthetic workloads (SW_1 and SW_2).

The synthetic workload (SW_1) is similar to the one used by the authors of Sparrow [23]. In the synthetic workload (SW_1), we use 2 independent clients to submit jobs. Each client submits a job at an interval of 10 ms, and each job consists of tasks with 10-ms durations. To vary the cluster utilization, we vary the number of tasks per job. For evaluating Bolt's task-constraint scheduling capabilities, we use another synthetic workload (SW_2) which has been described in further detail in Section 5.7.

Otherwise, unless specified, we use the Google cluster traces [45] in our evaluation. The Google traces include information for tasks running on a 12,500-node cluster at Google over a month. To generate a trace that we can run on our 12-node cluster in a manageable time, we followed a similar approach to that of Firmament [18] and Hawk [46]. We took a uniform sample of the trace and accelerated the sample to create a trace that completes execution on our cluster in 3 mins. We change the sampling rate depending on the number of executors in

the experiment. The resulting trace had a median task duration of 5 ms. Google's trace includes 12 levels of priority, while our implementation supports only four. We condensed the 12 priority levels in the trace into four levels. In all our experiments, we report the average of 10 runs. The standard deviation in all our experiments was under 5%.

In the rest of this chapter, we evaluate the throughput and latency of all the scheduling alternatives under a variety of conditions.

5.2 Scheduling Throughput and Scalability

Figure 11. Scheduling Throughput Comparison

Figure 11 shows the scheduling throughput of the different alternatives. For Sparrow, we evaluate its performance with 1 and 4 Sparrow schedulers. To increase the load on the schedulers, we use a synthetic benchmark composed of no-op tasks. We vary the number of executors to increase the load (x-axis in Figure 11). For Sparrow, we avoid the probing step in the protocol. This is a favorable configuration to Sparrow as the scheduler does not need to handle additional probing logic. Consequently, Sparrow's results show the upper bound for Sparrow's performance. For no-op tasks, an executor retrieves the task, immediately drops it, and requests a new task.

Figure 11 shows that Sparrow and Bolt-Server are not able to support large clusters. Bolt-Server and a single Sparrow scheduler achieve their highest throughput of 500 K decisions per second with 150 no-op executors. Using four Sparrow schedulers achieves up to 1.9 million scheduling decisions per second. Bolt's performance improves linearly with additional executors. With 720 executors Bolt performs 23 million scheduling operations, equivalent to over 45 Sparrow schedulers. Unfortunately, we could not deploy more executors on our cluster to stress Bolt.

When considering real-time workloads, the task execution times are higher than the no-op tasks used in this experiment. For example, in order to handle a cluster of 1000 executors executing 5 ms tasks at 100% utilization, Bolt would need to perform 200,000 scheduling operations per second. With 23 million scheduling operations per second, Bolt can already expect to handle a very large cluster of 115,000 executors. The switch data sheet indicates that the switch can handle up to 4.7 billion packets per second, indicating that Bolt's peak performance is significantly higher than the workload our no-op executors can generate, and represents millions of executors running 5 ms tasks.

5.3 Performance with a Synthetic Workload (SW_1)

For this experiment, we use the synthetic workload SW_1 described in Section 5.1.3. Jobs consist of tasks with 10-ms execution times. We deploy a total of 60 executors across a cluster of 10 machines, carrying out CPU-intensive integer arithmetic operations. This experiment tries to examine Bolt's performance in a hypothetical scenario with uniform job submission rates and task execution times.

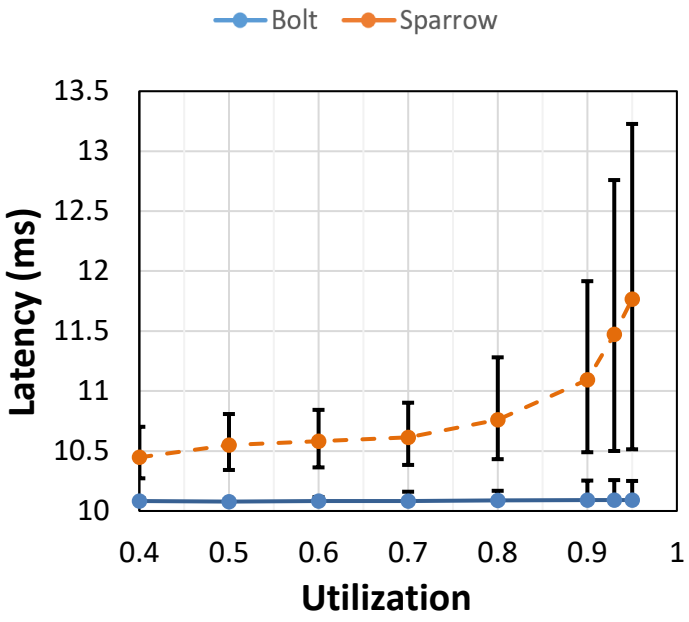


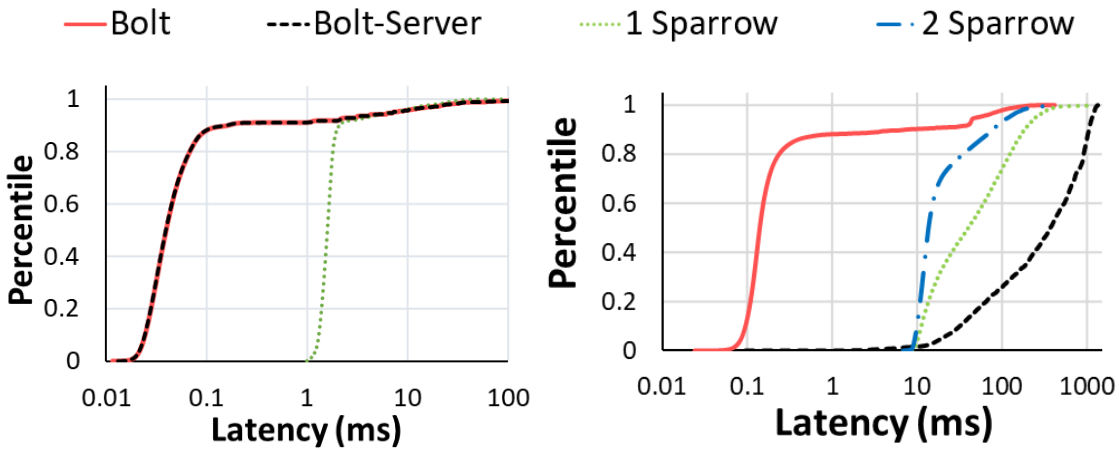
Figure 12. Job completion times for various cluster utilization levels with SW_1

Figure 12 shows the job completion time variations with increasing cluster utilization. We observe that as the cluster utilization increases, Sparrow's scheduling delays proportionally increase drastically. This is partly because higher cluster utilization means a larger number of tasks per job with 60 tasks per job at 100% utilization. At the 95% utilization rate, Bolt has a median scheduling delay 18x better than that of Sparrow (0.09 ms for Bolt vs 1.76 ms for Sparrow). Even at 50% cluster utilization, Bolt still performs 7x better than Sparrow. Unlike Sparrow, Bolt has a tighter range for the scheduling overheads as well. At the 95% utilization rate, we see that the error bars for Sparrow are quite large, indicating that the worst-case scheduling overhead is significantly larger than the median.

5.4 Performance with a Real Heterogenous Workload

Real-world workloads usually consist of bursty non-uniform job submissions with heterogenous task execution times. In order to examine such a realistic scenario, we evaluate the scheduling performance of Bolt, Bolt-Server and Sparrow using the Google Cluster Traces [45]. Figure 13.a shows the total scheduling delay of the different alternatives when we use the Google trace to run CPU-intensive tasks. Each task continually performs integer arithmetic

operations for the duration of the task. With this workload, each of the 11 worker nodes can run 16 executors (176 executors total). Figure 13.a shows that Bolt can reduce the scheduling delay by 40 times. While the median scheduling delay of a single Sparrow scheduler is 1.6 ms, Bolt's median delay is 0.04 ms. The main reasons for this performance difference are Sparrow's protocol and implementation overheads (detailed later in this section). Interestingly, our Bolt-Server implementation achieves comparable performance to Bolt. This is because the Bolt-Server implementation is lightweight and highly optimized. For Bolt and Bolt-Server, the scheduling delay is dominated by the network round trip time for the get_task() operation. We note that the latency increases to over 5 ms at the 95th percentile. This is due to the workload burstiness in which hundreds of tasks are submitted at the same time, leading to long tails in the queueing delay. We present a breakdown of this scheduling overhead in the following section.



(a) CPU-intensive workload (b) Sleep workload

Figure 13. Scheduling latency CDF. X-axes are in log-scale.

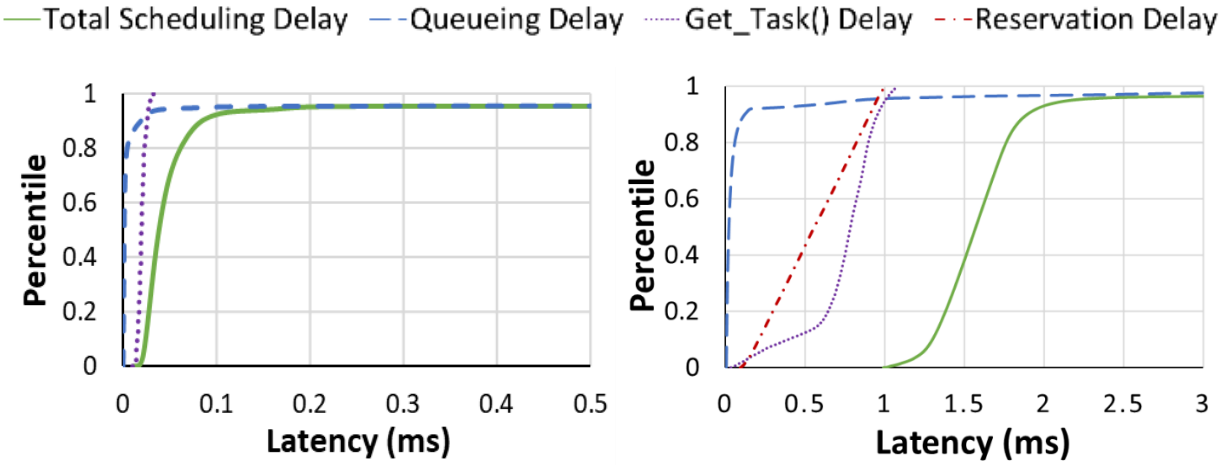
To generate a higher load, we change our executors to sleep for the duration of a task rather than performing arithmetic operations. With this change, we can run 200 executors per worker node. This configuration increases the scheduling workload by 12.5 times (2200 executors compared to 176 executors in Figure 13.a). Figure 13.b shows the total scheduling delay under this workload. Bolt reduces the scheduling delay by at least 100 times. While Bolt achieves a

median scheduling delay of 0.14 ms, all other alternatives achieve scheduling delays over 14 ms even when using 2 Sparrow schedulers (2 Sparrow in Figure 13.b).

Interestingly, Bolt-Server has a lower performance than a single Sparrow scheduler (370 ms vs 40 ms). This is because executors in Sparrow only send a get task request when they have a probe, but in Bolt-Server, executors continuously poll the scheduler for tasks. In our implementation, if a Bolt executor receives a no-op task, it waits for 50 μs before re-sending the get task request. While this configuration works well for Bolt, it generates high overhead for Bolt-Server.

5.5 Scheduling Overhead Breakdown

To understand the performance difference between the different designs, we measure the time spent on each step of the protocol. Figure 14 shows that CDF for each step of the protocol. The protocol steps are detailed in Figure 3 and Figure 5.b. Figure 14 shows the breakdown of the scheduling overhead for the experiment in Figure 13.a. The experiments use the Google trace with each task performing arithmetic operations for the duration of the task. The system uses 176 executors in total.



(a) Bolt breakdown (b) Sparrow breakdown

Figure 14. Scheduling overhead breakdown. Note that the x-axis is in a different scale in (a) and (b).

Figure 13.a shows that Bolt achieves 40 times lower median latency and 13 times lower latency at the 90th percentile. The two systems have a similar latency at the 95th percentile, because of the high queueing delay. The Google trace is bursty, with hundreds of tasks being submitted at certain times. These bursts of tasks are queued to wait for the next available executor.

Bolt's scheduling delay is dominated by the queueing delay (Figure 14.a). The get_task() delay equals the network round trip time for 90% of requests. For Sparrow, the scheduling delay is affected by two main factors. Figure 14.b shows that probing (the reservation step in Figure 3) is the reason for 37% of the total scheduling delay. The rest are get_task() and queueing delays. We dissected Sparrow's implementation to understand the source of the high

delay. We found that Sparrow uses separate threads for receiving requests and scheduling tasks. These threads communicate over a queue that is protected by a global lock. Similarly, a worker node running multiple executors has separate threads for processing packets and running tasks. These threads communicate over a queue that is protected by a global lock. The overhead of this implementation and thread contentions on the queues contributes to Sparrow's get task and reservation delays.

5.6 Priority-Based Scheduling

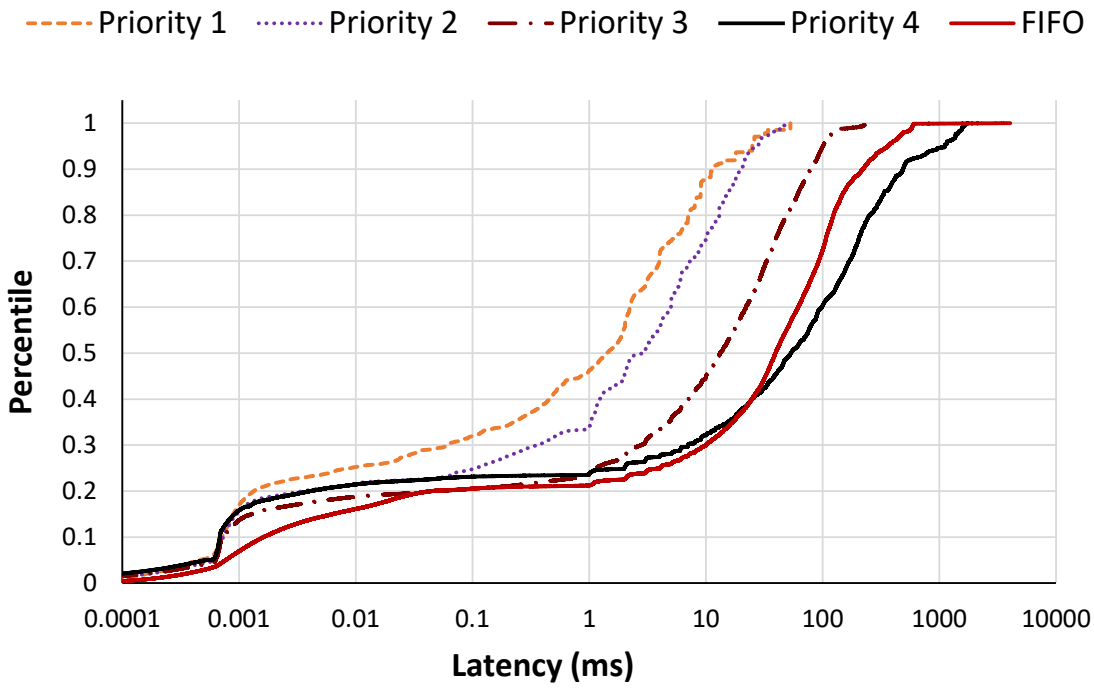


Figure 15. Queuing delays across different priority levels. The x-axis is in log-scale.

To demonstrate Bolt's priority-based scheduling, we run the CPU-intensive workload with 176 executors. The Google traces [45] have 12 levels of priority, while our implementation has four. We map every three levels of Google priorities to one priority level in Bolt. The resulting workload has 1.2%, 1.7%, 64.6%, and 32.2% of tasks at priority levels 1, 2, 3, and 4, respectively. Tasks at different priority levels experience different queueing delays, with higher priority tasks experiencing shorter queueing delays. To evaluate this effect, we increase the sampling rate to introduce longer queueing delays. Figure 15 shows the queueing delays of tasks at different priority levels. Tasks with priority levels 1, 2, 3 and 4 have median queueing

delays of 1.4 ms, 2.9 ms, 13.3 ms, and 53.5 ms, respectively. The same workload, without assigning priorities to tasks, (FIFO in Figure 15) has a median queuing delay of 39.5 ms. Priority level 1 (highest priority) tasks are only queued when there are no free executors to run them, which leads to the lowest queueing delay.

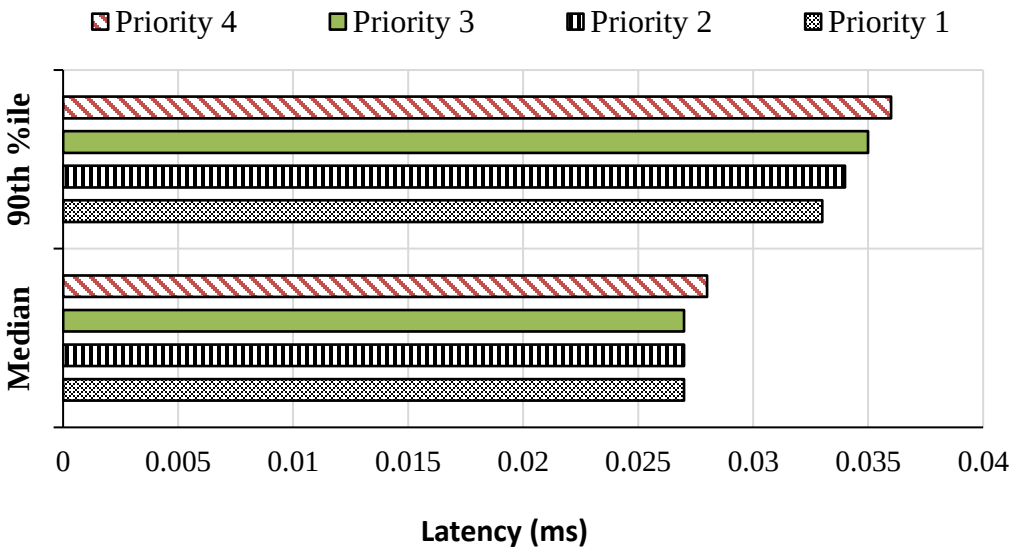


Figure 16. The delay of the get_task() operation at different priority levels.

Recirculation overhead. Bolt's priority-based scheduling relies on packet recirculation to check the task queues at different priority levels. Packet recirculation typically takes less than a microsecond. Figure 16 shows the latency of the get_task() step. The median latency of priority levels 1, 2, and 3 is statistically equivalent. The median latency of the get_task() operation for priority 4 is 2 μs higher than the other queues. The figure also shows the 90th percentile of the get_task() latency. The 90th percentile latency increases by 1-2 μs from one priority level to the next. These results show that the recirculation overhead imposed by Bolt's approach to supporting multiple priorities is negligible.

5.7 Scheduling with Resource Constraints

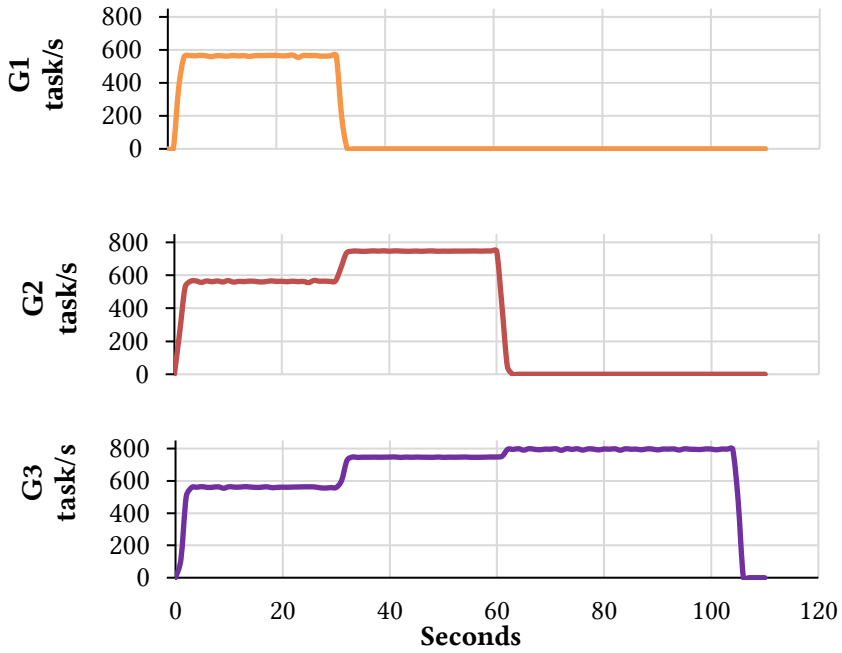


Figure 17. System throughput on a workload with node constraints (SW_2).

To demonstrate the effectiveness of scheduling with resource constraints, we design the following experiment. We assume that the cluster has three types of resources—A, B, and C—which represent different resources (e.g., GPU, large memory, hardware accelerators). We divide the cluster nodes into three groups: G1 has resource A, G2 has resources A and B, and G3 has resources A, B, and C. Tasks can specify the resources they need. For simplicity, we design a synthetic benchmark (SW_2) that has 20-ms tasks. Each task requests one of the three resources A, B, or C. Jobs have 90 tasks each and are submitted at 20-ms intervals.

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

Figure 17 shows the average throughput of a node from each one of the three node groups. In the first 30 seconds, all nodes in all groups are busy, as all nodes have the requested resource A. In the next 30 seconds, only the nodes in G2 and G3 are running tasks. In the last 30 seconds,

only G3 nodes are running tasks. We note that G3 nodes are overloaded. Thus, although the last task is submitted at the 90-s mark, the execution only finishes at the 110-s mark.

Chapter 6 – Related Work

6.1 Hybrid Scheduling

Hybrid Scheduling paradigms involve using a combination of centralized and decentralized scheduling techniques. The centralized and decentralized components usually work independently and on separate subsets of the submitted jobs. Jobs which require accurate task placement and tolerate higher scheduling delays are handled by the centralized scheduler component, while jobs requiring low latency scheduling are handled by the decentralized scheduler components. Unfortunately, these hybrid schedulers suffer the same drawbacks as their centralized and decentralized counterparts when handling real-time workloads. We will examine two examples of hybrid scheduling paradigms, Hawk [46] and Mercury [47] in this section.

6.1.1 Hybrid Scheduling with Hawk

Hawk [46] utilizes a hybrid scheduling model where long-running jobs are scheduled by the centralized scheduler, while short jobs are scheduled via a distributed approach, similar to that of Sparrow [23]. Hawk's [46] decentralized schedulers operate autonomously on the cluster while scheduling these jobs, with zero intercommunication between them. They rely on probing a subset of the nodes in the cluster to locate a node on which the short jobs can be scheduled. The drawbacks of such probing approaches have been discussed earlier in Chapter 2.

Hawk [46] differs from Sparrow [23] however, by additionally implementing random task stealing to compensate for the poor task placement by its decentralized schedulers. If a server is idle, it contacts a random set of other servers and selects one among them to steal tasks from. While this reduces head-of-the-line blocking compared to Sparrow [23], this approach still involves additional overhead compared to Bolt. The authors of Hawk [46] themselves note that the improvement gained by implementing random task stealing over Sparrow [23] is quite small for short jobs when running an accelerated version of the Google Cluster Traces [45].

Thus, Hawk [46] suffers the same drawbacks as Sparrow [23] and other traditional decentralized schedulers when running low-latency jobs.

6.1.2 Hybrid Scheduling with Mercury

Mercury [47] utilizes a hybrid paradigm as well. The Mercury-Runtime handles all job scheduling decisions and consists of both centralized and decentralized scheduling components. They also use pre-emptive cancellation of tasks in progress, to free up resources for other jobs depending on cluster conditions.

Mercury [47] utilizes a container-based model for task execution. Resources are allocated to jobs in the form of containers, and task execution happens via these containers as well. A job can choose between 2 kinds of containers upon submission; *Guaranteed* or *Queueable* containers. A *Guaranteed* container incurs no queuing delays and is guaranteed to run to completion i.e., without pre-emption. A *Queueable* container on the other hand, enables Mercury [47] to queue a task for execution on any node, without specific guarantees about task completion time or pre-emption mid-execution. *Guaranteed* containers, when allocated on a node already running a task from a *Queueable* container can cause the *Queueable* task to be pre-empted.

At first glance, it appears that scheduling short jobs without any head-of-line blocking can be accomplished simply by scheduling them via *Guaranteed* containers. However, *Guaranteed* containers can only be allocated by the centralized scheduling component within Mercury [47], As we have discussed in Chapter 2, traditional centralized schedulers do not have the high scheduling throughput needed to handle a large amount of low-latency jobs. This is reaffirmed by the authors of Mercury [47], acknowledging that performing all their scheduling decisions through their centralized scheduler would inadvertently cause it to be a performance bottleneck. Thus, Mercury's [47] hybrid scheduling paradigm suffers the same scalability concerns as traditional centralized schedulers when handling low-latency workloads.

6.2 Streaming Systems

The task-streaming model consists of (typically infinite) streams of tasks submitted to the system by clients. Stream processing can be handled by the *continuous operator* model, such as the one employed by Apache Flink [48], Naiad [49] and MapReduce Online [50], or a *bulk-synchronous-parallel* (BSP) model, used by Spark Streaming [51] and Drizzle [52].

In the continuous operator model [48, 49, 50] , the program is mapped into a sequence of long-running operators which are then deployed on nodes in the cluster. These operators only move between nodes upon node failures (via fault-tolerance mechanisms) but are otherwise statically tied to the node they are mapped to. Input records are sent sequentially to different nodes in a pre-constructed fashion based on the program dataflow. These systems have large fault-recovery times due to the fault-tolerance mechanisms they use. Although scheduling decisions need to be made with low overheads in such an environment, they are largely static, and do not change rapidly with differing cluster and workload conditions.

In order to improve fault-recovery times, Spark Streaming [51] and Drizzle [52] have resorted to using the bulk-synchronous-parallel (BSP) model, which is similar to the traditional Map-Reduce model. Here, records arrive at varying rates, and processing usually occurs by grouping them together at regular time intervals into mini-batches [51]. These tasks are then processed by a traditional system such as Spark [19] using in-memory Resilient-Distributed-Datasets (RDDs). However, due to the overheads associated with Spark described in Chapter 5, such systems typically have higher end-to-end latencies compared to continuous operator models. For instance, the authors of Spark Streaming [51] note that their system using D-Streams has an end-to-end latency of 500 ms – 2 seconds, compared to the 100-ms latency of their continuous operator counterparts. Drizzle [52] improves upon this to achieve an end-to-end latency of 350 ms. However, these latencies are still far higher than the average task execution times targeted by Bolt.

Additionally, the primary focus within all such streaming environments is to reduce the per-record latency experienced across a long sequence of streaming operators, as described by Li et al [33]. This is orthogonal to the focus of schedulers such as Bolt and Sparrow [23], which

target real-time jobs in a parallel compute environment, similar to the requirements of OLDI workloads [2, 3, 4, 5].

6.3 Network-Accelerated Systems

Numerous recent projects have used programmable switches in a wide range of applications including accelerating consensus protocols [38, 39, 53, 54], implementing in-network caching [55], for DNN training and inferencing [56], and for in-network aggregation operations [57]. However, none of these explore the usage of network-acceleration for scheduling decisions in parallel compute environments.

JumpGate [58] proposes a compilation and orchestration mechanism for in-network acceleration of data analytics functions by exposing an interface of primitives for usage by frameworks. The data-analytics framework submits the logical plan of operations (such as Filter and Shuffle) to JumpGate [58], which then proceeds to map these to stages of physical operators present on devices in the cluster. JumpGate [58] primarily focuses on providing aggregation-operation offloading capabilities as a service, and does not discuss the challenges associated with scheduling low-latency tasks.

R2P2 [59] proposes a scheduling approach for RPCs that leverages programmable switches. However, they do not maintain any tasks in-memory and simply recirculate an RPC call packet until a server is available to service the request. Thus, R2P2 [51] cannot be used to make the complex decisions associated with scheduling tasks in parallel-compute environments. Bolt stores tasks on switch memory until they are ready to be executed while supporting FIFO as well as complex task-priority based and resource-constraint based scheduling policies, thus making it a true scheduling system.

6.4 Low-Latency Efforts

Several projects over the years have explored operating system and network stack optimizations for low latency workloads. These efforts range from kernel-bypass techniques [60, 61, 62] to avoid the overheads of traversing the operating system’s network stack to efficient CPU core reallocation mechanisms [63]. However, these efforts are orthogonal to this

thesis as they focus on intra-machine delays while Bolt tackles the challenges associated with scheduling real-time workloads at cluster scale.

Chapter 7 – Conclusion and Future Work

Bolt is a novel in-network scheduling paradigm that overcomes the shortcomings of modern scheduling systems. It utilizes in-network acceleration via a programmable switch to ensure that it has the low scheduling overheads and high scheduling throughput needed to handle real-time applications on large clusters. Bolt can also support various scheduling policies such as FIFO, task priority aware and task resource constraint aware scheduling. Bolt offers these functionalities at a task granularity while most modern scheduling systems do so at a coarser job granularity. Most importantly, Bolt can handle these policies without sacrificing performance.

In the future, we plan to evaluate Bolt against scheduling protocols implemented using kernel bypass mechanisms such as the Intel Data Plane Development Kit (DPDK) [64]. We would also like to compare Bolt against network-accelerated systems such as R2P2 [59]. We are exploring other frontiers to evaluate the pros and cons of push-based vs pull-based scheduler designs on programmable switches as well.

Bibliography

[1] Ibrahim Kettaneh, Sreeharsha Udayashankar, Ashraf Abdel-hadi, Robin Grosman, and Samer Al-Kiswany. 2020. Falcon: Low Latency, Network-Accelerated Scheduling. In *Proceedings of the 3rd P4 Workshop in Europe*. Association for Computing Machinery, New York, NY, USA, 7–12.

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

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

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

[5] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2: 74–80.

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

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

[8] S. Verma, Y. Kawamoto, Z. M. Fadlullah, H. Nishiyama, and N. Kato. 2017. A Survey on Network Methodologies for Real-Time Analytics of Massive IoT Data and Open Research Issues. *IEEE Communications Surveys Tutorials* 19, 3: 1457–1477.

[9] Stephen F Elston and Melinda J Wilson. Big Data and Smart Trading. Retrieved from https://dsimg.ubm-us.net/envelope/86703/367082/1348004765_SYB_Big_Data_and_Smart_Trading_WP_Apr2012_WEB.pdf.

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

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

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

[13] Ciamac Moallemi and Mehmet Saglam. 2013. OR Forum—The Cost of Latency in High-Frequency Trading. *Operations Research* 61, 5: 1070–1086.

[14] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. 293–307.

[15] Rajesh Nishtala, Hans Fugal, Steven Grimm, et al. 2013. Scaling Memcache at Facebook. 385–398.

[16] John Ousterhout, Arjun Gopalan, Ashish Gupta, et al. 2015. The RAMCloud Storage System. *ACM Transactions on Computer Systems* 33, 3: 7:1-7:55.

[17] W. Chen, A. Pi, S. Wang, and X. Zhou. 2018. Characterizing Scheduling Delay for Low-Latency Data Analytics Workloads. *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 630–639.

[18] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*.

[19] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10–10: 95.

[20] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, USENIX Association, 137--150.

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

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

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

[24] Tofino-2 Second-generation of World's fastest P4-programmable Ethernet switch ASICs. Retrieved March 16, 2020 from https://www.barefootnetworks.com/products/brief-tofino-2/.

[25] Tofino World's fastest P4-programmable Ethernet switch ASICs. Retrieved March 16, 2020 from https://www.barefootnetworks.com/products/brief-tofino/.

[26] Trident3-X7 / BCM56870 Series. Retrieved January 21, 2021 from https://www.broadcom.com/products/ethernet-connectivity/switch-fabric/bcm56870.

[27] Luiz Andre Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro* 23: 22–28.

[28] Panagiotis D. Diamantoulakis, Vasileios M. Kapinas, and George K. Karagiannidis. 2015. Big Data Analytics for Dynamic Energy Management in Smart Grids. *Big Data Res.* 2, 3: 94–101.

[29] Y. Yan, Y. Qian, H. Sharif, and D. Tipper. 2013. A Survey on Smart Grid Communication Infrastructures: Motivations, Requirements and Challenges. *IEEE Communications Surveys Tutorials* 15, 1: 5–20.

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

[31] O. Elijah, T. A. Rahman, I. Orikumhi, C. Y. Leow, and M. N. Hindia. 2018. An Overview of Internet of Things (IoT) and Data Analytics in Agriculture: Benefits and Challenges. *IEEE Internet of Things Journal* 5, 5: 3758–3773.

[32] Philipp Moritz, Robert Nishihara, Stephanie Wang, et al. 2018. Ray: A Distributed Framework for Emerging AI Applications. *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, USENIX Association, 561–577.

[33] Boduo Li, Yanlei Diao, and Prashant Shenoy. 2015. Supporting Scalable Analytics with Latency Constraints. *Proc. VLDB Endow.* 8, 11: 1166–1177.

[34] N. Tolia, D. G. Andersen, and M. Satyanarayanan. 2006. Quantifying interactive user experience on thin clients. *Computer* 39, 3: 46–52.

[35] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices. *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, Association for Computing Machinery, 155–168.

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

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

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

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

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

[41] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*.

[42] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, et al. 2013. Apache Hadoop YARN: yet another resource negotiator. *Proceedings of the 4th annual Symposium on Cloud Computing*, Association for Computing Machinery, 1–16.

[43] Benjamin Hindman, Andy Konwinski, Matei Zaharia, et al. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, USENIX Association.

[44] 2013. Sparrow Git Repository. Retrieved January 23, 2021 from https://github.com/radlab/sparrow.

[45] John Wilkes. Google ClusterData 2011 Traces. *GitHub*. Retrieved January 16, 2021 from https://github.com/google/cluster-data.

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

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

[48] Paris Carbone, Asterios Katsifodimos, † Kth, et al. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Engineering Bulletin* 38.

[49] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, Association for Computing Machinery, 439–455.

[50] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce Online. *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association, 21.

[51] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, Association for Computing Machinery, 423–438.

[52] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, et al. 2017. Drizzle: Fast and adaptable stream processing at scale. 374–389.

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

[54] Hatem Takruri, Ibrahim Kettaneh, Ahmed Alquraan, and Samer Al-Kiswany. 2020. FLAIR: Accelerating Reads with Consistency-Aware Network Routing. *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 723–737.

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

[56] Dan RK Ports and Jacob Nelson. 2019. When Should The Network Be The Computer? *Proceedings of the Workshop on Hot Topics in Operating Systems*, 209–215.

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

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

[59] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. 2019. R2P2: Making RPCs first-class datacenter citizens. *2019 USENIX Annual Technical Conference (ATC 19)*.

[60] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. *Proceedings of the 2014 ACM Conference on SIGCOMM*, Association for Computing Machinery, 175–186.

[61] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. *Proceedings of the 26th Symposium on Operating Systems Principles*, Association for Computing Machinery, 325–341.

[62] Dominik Scholz. 2014. A Look at Intel's Dataplane Development Kit. .

[63] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association, 361–377.

[64] DPDK. *DPDK*. Retrieved January 18, 2021 from https://www.dpdk.org/.