Q-Flink: A QoS-Aware Controller for Apache Flink

M.Reza HoseinyFarahabady‡, Ali Jannesari†, Javid Taheri¶, Wei Bao∗, Albert Y. Zomaya∗, Zahir Tari†
†The University of Sydney, School of Computer Science,
‡Department of Mathematics and Computer Science, Karlstad University, Sweden
¶RMIT University, School of Science, Australia
∗The University of Sydney, School of Computer Science, Australia
DOI 10.1109/CCGrid49817.2020.00-30
978-1-7281-6095-5/20/$31.00 ©2020 IEEE

Abstract—Modern stream-data processing platforms are required to execute processing pipelines over high-volume, yet high-velocity, datasets under tight latency constraints. Apache Flink has emerged as an important new technology of large-scale platform that can distribute processing over a large number of computing nodes in a cluster (i.e., scale-out processing). Flink allows application developers to design and execute queries over continuous raw inputs to analyze a large amount of streaming data in a parallel and distributed fashion. To increase the throughput of computing resources in stream processing platforms, a service provider might be tempted to use a consolidation strategy to pack as many processing applications as possible on the working nodes, with the hope of increasing the total revenue by improving the overall resource utilization. However, there is a hidden trap for achieving such a higher throughput solely by relying on an interference-oblivious consolidation strategy. In practice, collocated applications in a shared platform can fiercely compete with each others for obtaining the capacity of shared resources (e.g., cache and memory bandwidth) which in turn can lead to a severe performance degradation for all consolidated workloads.

This paper addresses the shared resource contention problem associated with the auto-resource controlling mechanism of Apache Flink engine running across a distributed cluster. A controlling strategy is proposed to handle scenarios in which stream processing applications may have different quality of service (QoS) requirements while the resource interference is considered as the key performance-limiting parameter. The performance evaluation is carried out by comparing the proposed controller with the default Flink resource allocation strategy in a testbed cluster with total 32 Intel Xeon cores under different workload traffic with up to 4000 streaming applications chosen from various benchmarking tools. Experimental results demonstrate that the proposed controller can successfully decrease the average latency of high priority applications by 223% during the burst traffic while maintaining the required QoS enforcement levels.

Index Terms—Massive Data Stream Processing, Apache Flink, Meta-Scheduling, Resource Allocation, Computer System Modeling and Profiling, Model Predictive Controller

I. INTRODUCTION

Rising demand for inter-connectivity in recent years and the abundance of emitted data have had tremendous impact while introduced new challenges and greater opportunities for companies in information and communications technology space. The abundance of data in nearly every imaginable industrial domain have had tremendous impact while introduced new challenges and greater opportunities for companies in information and communications technology (ICT) space. To extract the best value out of the collected raw data, one needs to perform a refinement process (which often include cleanse, standardize, and transformation steps) to create more applicable information. Nevertheless, any effort to run the analytical process in a near real-time fashion can dramatically increases the economic value of such raw data, too.

The idea of processing data-in-motion initially started with traditional database management systems, which were capable for executing continuous query over streamlined data [1], and then elaborated with the advent of complex event processing (CEP) systems [2] later on. Many businesses have already adapted a wide variety of distributed data analytics technologies with the main aim of achieving a real-time big data processing for reaching to a higher level of customer satisfaction. Exceptional growth of streamline analytic engines over the last few years, including MapReduce [3], Apache Storm [4], Twitter Heron [5], IBM Streams [6], Google MillWheel [7], Oracle Stream Analytics [8], Microsoft Sonora [9], Apache Spark [10], Azure Stream Analytics [11], and Apache Flink [12] can truly reflect the importance and potential growth of big data processing systems in our modern society. The idea of executing continuous query over streamlined data [1] is elaborated with the advent of complex event processing (CEP) systems [2]. More recently, the concept of data transformation pipelines has received considerable attention owing to their great potential for crafting a strategic plan while maintaining the critical low-latency requirements that need to update data products in the order of (×10²) milliseconds.

Comparatively, such systems often provides developer team with a set of high-level application programming interfaces (API) which make designing and developing highly complex SQL-like operations faster and easier. However, it has not been possible till recent advances on distributed systems to exploit a platform with capability to span such functionalities at scale, i.e., a cluster with tens or hundreds of nodes. Particularly, the emergence of analytical streaming applications with low-level millisecond latency requirements – e.g., those derived in the context of Internet of Things (IoT) eco-systems– calls for innovative design solutions to deploy such computations on multiple computing devices. Stream-data processing promotes a new, yet important concept across contemporary industries, where success or failure of a decision is solely determined by the response time to correctly react to an external event incident [13]–[16].

One of the basic for modern stream processing infrastruc-
tures is their ability to process a large volume of data flows in a real-time and in a scalable manner [17], [18]. To process a large volume of data flows in a real-time and in a scalable manner, taking advantage of a distributed cluster consisting of tens or even hundreds of server nodes seems inevitable [19]. Virtualization technologies along with affordable multi- and many-core chips can be efficiently exploited to successfully build such infrastructures [13]. On the other side, stream data processing engines usually endure bursty traffic with low frequency but high intensity, particularly under special events like big sales [20]. Several past studies confirm the existence of load spikes with varying burst intensities and durations in a routine data processing platform. However, the processing capacity of working nodes in a data center platform is limited which can result in violation of the quality of service (QoS) performance target under the load spikes. While satisfying the QoS level of latency-critical data processing applications plays a pivotal role in the financial success of a business, it is vital to strictly fulfill the QoS enforcement level under such traffic burst conditions [21].

Nevertheless, in a shared environment where multiple applications want to access limited computing resources (CPU, Memory, Disk and Network), a static resource allocation mechanism can result in over/under-commitment of resources and wasting of investment cost [22], [23]. A static resource manager, which assigns a fixed number of computing resources to each data-stream application, can disappointingly result in under/over-provisioning of resources, particularly when a sudden change in the incoming traffic of some streams occurs. An effective resource management policy can guarantee more predictable service rates as the demand and capacity vary over time, while preventing performance degradation when several applications compete with each other to acquire shared resource capacity (e.g., CPU cache and memory read/write bandwidth) [24]–[26]. One of the well-known examples is the so-called micro-architecture slowdown that occurs when several collocated applications cause a huge increase in the overall memory access latency by evicting the data belonging to each others residing in the last level cache [27]. To the best of our knowledge, none of the commercial resource allocation strategies developed for large-scale data-streaming processing engines is aware of such interference at the micro-architecture level.

In this paper, we propose Q-Flink, a QoS-aware feedback controller system that adaptively adjusts the CPU cap and memory buffer size for latency-critical applications in a Flink platform. By continuously monitoring the workload patterns and using model predictive control (MPC) design principles, Q-Flink can predict the load intensity in the next sampling interval and to rapidly respond to workload fluctuations by elastic allocation of computing resources to comply with the given QoS requirements, e.g., the accepted upper limit for $p$-$99$ latency of each query. Linux container (LXC) is an operating-system-level virtualization technique for running multiple isolated applications on a server node using a single Linux kernel [28]. The proposed approach uses model predictive control (MPC) design principles when it hosts several Flink applications. Each application might have its own quality of service (QoS), e.g., as expressed by $p$-$99$ latency to be experienced by each query.

We exploit this fact that at high traffic loads, the end-to-end response time is mainly determined by the waiting time of data tuples in the corresponding queues. We propose a performance model to predict the latency of each query in the run-time. Using such a model, the controller can reduce the end-to-end response time which is mainly determined by the waiting time of data tuples in the corresponding queues during high traffic loads, hence, reducing the QoS violation rate. It is also equipped with a feedback control loop to monitor the shared-resource interference among collocated applications in response to the continuous feedback from the performance state of the underlying resource usages. Experimental results confirm that the proposed solution outperforms the default Flink scheduling policy under all non-burst and burst workload conditions. Particularly, the proposed controller can decrease the $p$-$99$ latency of applications in the highest QoS class by $139\%$ during burst periods in comparison with the output of Flink’s default policy.

Section II motivates the problem of designing a new resource allocation control in high-throughput stream processing platforms under sudden workload surges. Section III formalizes the research question. Section IV gives insights into the proposed controller. Section V summarizes the experimental evaluation results, followed by our conclusions in Section VII. Finally, In the Appendix Section, we provide the reader with essential background needed to better understand the stream processing model provided by Apache Flink.

II. MOTIVATION

There are several statistical reports that confirm the diversity of workload patterns and the QoS requirements of stream and batch data processing applications in clusters with shared resources increases [29], [30]. However, simple resource allocation policies, such as round-robin, which are commonly found in most of data processing engines, can not effectively cope with the surges in the incoming workload or QoS violations during the run-time. To further illustrate the importance of equipping the default resource allocation policy with a controlling mechanism to dynamically react to the run-time conditions in a shared environment, we conducted a simple experimental setting with 60 applications in the platform, each contains a non-trivial stream processing logic that is designed to read ordered log files, and then performs analytical computations as the events are received. This section highlights the negative performance impact of shared resource interference among collocated queries when they run over a cluster with four worker nodes.

Each application runs a specific type of operation (query) over its input data (working hours of NY taxi drivers) and calculates the breaking time of individuals within a predefined window. We classify applications into three different QoS classes, denoted by $g_{1-3}$. Figure 1 shows the impact of default
Flink’s scheduling policy on the achieved performance level of applications in different QoS classes.

![Graph showing p-99 response time of 60 stream processing applications with different QoS requirements. During the burst period, the incoming traffic rate of each application in q3 suddenly rises from 100 to 300 items per second.]

The figure shows how unsatisfactorily the default policy affects the p-99 response time of high priority applications (q1). During the burst period, the results confirm that the analytical applications in q1 or q2 classes lack of required computing resource to satisfy the target p-99 response time. Queued workloads trigger a longer waiting time that consequently leads to a higher query response time; hence, causing higher QoS violation incident rates for high priority applications.

III. PROBLEM STATEMENT

The performance evaluation outcomes in a distributed stream data processing cluster confirm the inflexibility of a round-robin resource allocation algorithm to respond to temporal performance variations. In practice, a round-robin policy is dependent upon the arrival time of incoming streaming data to prioritize different requests; hence, it shows a poor performance to fulfill the complex quality of service enforcement level imposed by different end-users. The problem needs to be more carefully addressed in a distributed processing platform that computing resources are shared among multiple applications with different QoS requirements. As explained in previous sections, the experimental results confirm that even a more advanced solution (e.g., Weighted Fair queuing and Class-Based Weighted Fair Queuing policies) could cause a significant performance degradation, leading to a massive reduction in the profitability of the service provider. The other solutions proposed for general batch- or stream-processing platforms can impose higher run-time overheads making their implementation an impractical choice.

**Shared Resource Interference:** Finding an effective consolidation deployment plan in a platform with shared resources in the run-time is challenging, due to the fact that each data-processing application could have its own resource utilization characteristic which highly depends on the incoming traffic rate, different QoS enforcement, and sensitiveness to the available shared resources (e.g., last level cache size). The contention among consolidated applications to obtain the capacity of shared resources is an important factor that makes it difficult to design an effective consolidation deployment plan. Such incident has been reported for other parallel distributed platforms, e.g., see [24], [25], [27], [31]–[36] but to the best of our knowledge, no systematic empirical research exists to prevent the negative impact of shared resource contention in a Flink platform.

The aim of this paper is to tackle the problem of distributed flow control for high-speed query processing in Apache Flink, as a low-latency streaming processing platform, by dynamically allocating CPU and memory capacity to each application based on the run-time conditions and QoS violation incidents. Particularly, we assume that there are exactly Q different QoS classes that an application owner can select from and is billed accordingly. We identify a QoS class as a value pair, denoted by \((r^*_q,m,V_{q,Δt})\), where \(r^*_q,m\) is the maximum delay that applications in class \(q\) can tolerate before collecting the result of the corresponding query. If the delay of query processing takes longer than \(r^*_q\), this will be labeled as a QoS violation incident [26]. The end-to-end response time of a query is mostly the sum of two major parts of queuing delay (in the corresponding buffers) and the operational time (due to the internal structure of the underlying engine) [16]. Likewise, \(V_{q,Δt}\) denotes an upper bound for the percentage of QoS violation incidents, seen by applications in class \(q\), during any interval of length \(Δt\).

The proposed solution deals with those run-time states in which the queuing waiting time of the corresponding QoS class for each query is at least one order of magnitude higher than the operational processing time of the query. The primary aim is to maximize the weighted sum of the QoS satisfaction, which is equivalent to minimize the weighted QoS violation incidents, among all submitted queries belonging to different streaming applications. To deal with the increasing complexity of large-scale platforms, we designed a non-centralized controlling mechanism to find a near optimal solution among working nodes, while imposing low overheads for monitoring the run-time performance level of resources and applications.

IV. THE DYNAMIC RESOURCE ALLOCATION CONTROLLER

This section provides details of the core modules and algorithms of the proposed solution as a non-centralized resource allocation controlling mechanism for a Flink platform.

A. System Design Consideration

The three main design considerations of the proposed solution can be summarized as follows: (i) to achieve a decentralized architecture, in which the resource allocation decisions (e.g., the controlling actions) are made solely based on local information gathered in each working node; (ii) to impose only...
a very small overhead to the monitored application for online performance monitoring; and finally (iii) to augment the target platform as a separate proxy layer. The proxy architecture enables the controller to receive data streams from various data generators, while dispatching the streaming results to different kinds of data sinks. The embedded CPU and memory cap mechanisms do not interfere with the internal operational mechanisms of Flink or JVM CPU/memory management units. Such a separation allows the controlling layer to be easily activated/deactivated based on the internal service provider policies. This will facilitate the implementation of the controlling approach as well as provide a more elastic solution as its scalability limit is becoming entirely detached from the barrier of the Flink’s scalability.

B. Solution Overview

The proposed solution is based on a model predictive control (MPC) theory that uses run-time throttling over CPU and memory resources per submitted application to prevent each working nodes becoming a bottleneck and also to resist against the temporal changes in the arrival rate of each streaming flow. The idea is to postpone making the CPU cap and memory allocation decision until the execution time, in which the controller can infer appropriate information, including the workload arrival rate, the isolated CPU/memory resource availability, the QoS violation rate per application/query, the amount of shared resource contention, and the resulted performance degradation. The proposed MPC controller is based on a mathematically well-defined control design approach that firstly predicts the future behavior of the involving parameters (e.g., the incoming traffic rate per stream flows), and then dynamically makes the resource adjustment decisions based on both the current and the predicated future states of every working node. One of the advantages of using MPC approach is its robust performance against any modeling error by considering the present, the past, and the future of the system states [37].

Figure 2 depicts Q-Flink architectural overview along with the proposed core components. To comply with the design consideration points (Section IV-A), each streaming application is running within its own isolated container across the distributed platform. The Linux kernel provides a lightweight operating-system-level virtualization method for running multiple applications (the Linux container API or LXC) on a control host that allows limitation and prioritization of CPU, memory, and block I/O resources for applications. Control group (cgroups) is the essential kernel feature that controls and limits the resource usage for a group of isolated processes while they can share a single operating system kernel and are isolated from the rest of the system. As a result, a collection of services belonging to different clients can run in parallel on a single operating system kernel. In the proposed solution, we assign one MPC-based controller per each working node. At each controlling instant $\tau \in \{t_1, t_1 + \Delta t, t_1 + 2\Delta t, \cdots\}$, every host controller compares the sampled performance metric of each submitted processing applications to the desired absolute end-to-end delay of each applications. Based on the error values, the local controller decides about the new processing/memory budget of every worker process by adjusting $cpu\_share$ and $mem\_share$ parameters for the the corresponding cgroup. The main aim of the controller is to reduce the measured error to zero in the future controlling steps.

C. Main Components

The proposed architecture comprised of the following four main components: queuing model, incoming rate forecaster, shared resource anti-saturation, and CPU and memory cap optimizer. The queuing model provides an abstract system model for the dynamic estimation of the queuing waiting time of each stream in the main buffer per streaming query in the execution time. The forecaster provides an estimation of the input traffic rate of each streaming buffer for the next controlling intervals using past collected statistics. The anti-saturation module prevents over-utilization of shared resources when their usage is beyond a normal threshold, which indicates a presence of a fierce competition among consolidated workloads resulting in severe performance degradations. Lastly, the optimizer module continuously adjusts the amount of CPU and memory cap per streaming application while considering the future states of the system and the QoS violation incidents.

Particularly, let $y_*$ and $r$ denote the measured and ideal value (set-point trajectory vectors) for the target performance metric of a given application during an arbitrary controlling interval $\tau$. We allow the controller to select a controlling action (by adjusting the CPU and memory cap) such that the measured performance $y_*$ converges toward the set-point trajectory $r$ over the next $T_{ref}$ intervals at an exponential rate. The $T_{ref}$ value is referred to as “response speed factor”, and is a system design choice to be set by system administrator. The higher value for this parameter often results in less over- or under-shooting around the target output (i.e., a higher tolerance against errors in the system model or in the prediction module) while increases the settling time of the controlling actions (i.e., the time elapsed from the performance monitoring of an application to the time at which each performance output has entered and remained within a specified error band from the ideal value).
D. State Independent Performance Model

This section describes a performance model, namely an algebraic relationship among monitored parameters, for predicting the end-to-end response time of processing the data items coming to each stream flow. We use Allen-Cunneen (A-C) approximation of $G/G/N$ queue [38] to estimate an upper-bound for the mean end-to-end response time (i.e., the sum of the observation in the main buffer of the proxy-tier and the actual processing time) as experienced by each data item,

$$W_M = \frac{P_{bk,N}}{\mu N(1 - \rho)} \left( \frac{C_S^2 + C_B^2}{2} \right)$$  \hspace{1cm} (1)

where $W_M$ denotes the average waiting time of customers in a $G/G/N$ queue consists of $N$ working servers with a general distribution of both arrival time and service time, $\rho$ is the service traffic intensity (server utilization), $C_D = \sigma_D/E_D$ is the coefficient of variation for inter-arrival time, $C_S = \sigma_S/E_S$ is the coefficient of variation for service time, $\lambda^2 + \epsilon^2$ term is the stochastic variability of the queue, and $P_{bk,N}$ represents the probability that all workers in the queuing system are busy (i.e., the waiting time of a new customer is greater than zero).

E. Prediction Module

The auto regressive integrated moving average (ARIMA) model [39] is used here to estimate the arrival tuple rate, as a non-controllable parameter at any sampling interval $\tau$. To design the controller in the most general form, we assume that the probability distribution of the arrival rate of data tuples toward each streaming application is not known a priori. Hence, we develop a mechanism to estimate the future values of such a parameter by applying a stochastic analysis over a series of previous $h$ observations as follows:

$$\lambda_\tau = \epsilon_\tau + \sum_{\ell=1}^{h} \beta_\ell \lambda_{\tau-\ell} + \theta_\ell \epsilon_{\tau-\ell},$$

where $\lambda_{\tau-\ell}$ is estimated arrival rate, $\epsilon_\ell$ are independent and identically distributed (i.i.d) random errors from a normal distribution with mean zero and a finite variance, $\beta_\ell$’s and $\theta_\ell$’s are coefficients to be updated using the least-squares regression method right after each new observation.

F. Anti-Saturation

To cope with the performance degradation problem caused by severe contention among consolidated workloads to obtain shared CPU cache or memory bandwidth, we pursue an effective method proposed in [27] that appropriately quantifies the performance slowdown due to micro-architecture interference. This method identifies the workloads contention by identifying any abnormal increase in the memory bandwidth utilization of the target working node, which is calculated by monitoring two standard hardware events of $\text{UNC\_QMC\_READS}$ (an indicator of memory reads) and $\text{UNC\_QMC\_WRITES}$ (an indicator of memory writes).

G. Model Predictive Control Approach

Model Predictive Control (MPC) is a well-established controlling mechanism that uses prediction, system mode, and optimization of a dynamic system to force a set of particular output parameters follow specified target values while minimizing the output error from the set-point trajectories. At each controlling epoch $\tau = 1, 2, \cdots$, the MPC module in each working node makes resource allocation decisions (as a set of controlling action) as a sequence of the following actions:

- The profiler measures the number of pending data items in the associated buffer of each streaming applications to estimate its mean response time in the future $\kappa$ epochs by applying Equation (1) as well as the estimation tool (e.g., the ARIMA model).
- The profiler captures the degree of shared resource interference among collocated applications in each host by capturing the above-mentioned two hardware events.
- The controller computes the desirable CPU cap and buffer size to every streaming applications such that the corresponding response time in the future $\kappa$ epochs meet the QoS enforcement level.
- The optimization module determines the possibility of fulfilling all CPU and memory buffer demands requested by all streaming applications in a single host. In case of resource scarcity, the optimization module maximizes a reward function using a *cost-benefit* analysis which prioritizes demands coming from applications with the highest QoS enforcement level.
- Once the resource size for each application is determined, the controller applies just one step controlling action with regards to the response speed factor, i.e., $T_{ref}$ (Section IV-C). At the next controlling epoch $\tau + 1$, the controller re-monitors the platform run-time characteristics as a feedback loop and the whole cycle of modeling, forecasting and the optimization process is repeated.

H. Optimization Process

The controller needs to determine a partial fulfillment of CPU cap and buffer size for each application if the entire demand is higher than the available resource capacity in a working node. Let $C_{S,\tau}$ denote the amount of CPU demanded by a stream processing application $S$ at epoch $\tau$, and $R$ denote the computing capacity which is available in a particular working node. We define a contribution function for a partial fulfillment of the requested CPU to each stream $S$ as

$$\mathcal{C}(r_S) = I_{q_S} \times (r_S - C^*_S),$$  \hspace{1cm} (2)

where $r_S$ is the amount of CPU to be allocated to $S$ at the next epoch, and $q_S$ is the QoS class of $S$, and $I$ is an associated constant weight represent the importance of each $q_S$. The optimization module will maximize the total contribution received by service provider as follows

$$\max_r \sum_{S \in \mathcal{S}} \mathcal{C}(r_S),$$  \hspace{1cm} (3)

subject to the constraints of $r_S \leq C^*_S$, $\forall S \in \mathcal{S}$, and $\sum_{S \in \mathcal{S}} r_S = R$, where $\mathcal{S}$ represents the set of streaming applications assigned to be run by Apache Flink in the working
machine. A dynamic programming approach is used to compute a fast approximation solution for the above optimization problem. By recursively solving the following Bellman’s equation, the controller can find an optimal solution for Equation (3) after the discretization process:

\[ V_\omega(R_\omega) = \max_{0 \leq r_\omega \leq R_\omega} V_{\omega+1}(R_\omega - r_\omega) + C(r_\omega), \]

where \( V_\omega(.) \) denotes the optimal reward of allocating \( R_\omega \) resources among all non-allocated applications with the initial value of \( V_{\omega=|S|}(R) = \max_{0 \leq r_S \leq R} C(r_S) \).

V. EXPERIMENTAL PERFORMANCE EVALUATION

The proposed controlling system is implemented using C programming language. This system has been integrated with Apache Flink engine as its buffer management tool (see Figure 2).

A. Experimental Setup & Benchmark Suite

The hardware platform we used consists of four worker nodes. Each computing node runs a 64-bit Linux 5.0-generic kernel and is equipped with eight Intel Xeon cores, 16GB RAM, and total 256 GB SCSI-v3 storage drive. We implemented and deployed analytical applications from the following benchmark suits to represent a variety of application domains.

- A data pipeline (DPIP) benchmark using distributed pub-sub messaging system to consume large text streams generated from a synthetic data load generator (e.g., from a public data set for NY taxi rides) as described in [40]. The benchmark reflects the characteristics of stream analytical applications sending read- and write- heavy load query patterns.
- Grep, Sort, and MD5 (GSM) micro-benchmarks in BigDataBench 4.0 running on randomly generated streaming data. We choose the mentioned three micro-benchmarks as they are the most fundamental operations frequently used in various domains in data analysis; for example, Grep extracts matching strings from a set of plain data and returns the number of matching the input strings. BigDataBench is an open-source suite representing real-world data sets and large-scale data analytics workloads [41].

We run three scenarios for each application by setting the corresponding incoming data rate to \( \lambda \in \{100, 400, 1600\} \) tuples per second. We referred to such scenarios as low, middle and high traffic periods, respectively. In addition, each workload is randomly assigned to one of the three predefined QoS classes, denoted by \( q_{i=1...3} \), in which \( q_1 \) represents the set of applications with the most stringent latency requirements.

B. Result Summary

1. Latency & Throughput: Figure 3 shows the applications’ performance degradation in different QoS classes with regards to \( p\text{-}99 \) latency metric during different traffic conditions. Particularly, the improvements in \( p\text{-}99 \) latency during high workloads for \( q_1 \) and \( q_2 \) classes are 260% and 40%, respectively. The experimental results confirm that the proposed controller can reduce the \( p\text{-}99 \) latency and its variance under different traffic conditions for different benchmark suits that also meets the QoS requirements.

2. QoS violation: Figure 5 shows the ratio of QoS violation compared to the default behaviour of Apache Flink for varying incoming traffic rates. The target \( p\text{-}99 \) latency for applications in \( q_1 \) and \( q_2 \) are set to 200 and 450 milliseconds for DPIP and 300 and 500 milliseconds for GSM scenarios, respectively.

Fig. 3. Comparison of \( p\text{-}99 \) latency achieved by the proposed controller compared to the default behaviour of Apache Flink for varying incoming traffic rates. The target \( p\text{-}99 \) latency for applications in \( q_1 \) and \( q_2 \) are set to 200 and 450 milliseconds for DPIP and 300 and 500 milliseconds for GSM scenarios, respectively.

Fig. 4. Mean throughput, i.e., number of accomplished operations per sec. under non-burst and burst periods \( (\tau = 4000) \). Controller employs an anti-saturation mechanism to keep the stability of CPU usage in a safe range while fulfilling the QoS requirements.
compared to the results obtained by applying the default scheduler for the GSM benchmark. Such a metric directly reflects how well a policy can satisfy the requested service level agreement. We set the corresponding weight factor of each QoS class, i.e., $I(q_i = 1 \cdots 3) = \{6, 3, 1\}$ in Eq. (2) and the target $p$-99 latency to $\{300, 500, 800\}$ ms for different $q_i$'s, respectively. Using the cost-benefit analysis, the proposed controller can dynamically identify the resource shortage and reduce the overall QoS violation rates for higher priority applications.

3. Sensitivity analysis: We conducted additional experiments to measure the sensitivity and robustness of the controller with regards to the accuracy in the prediction model, particularly if there are inaccuracies in the estimation model. We conducted the sensitivity analysis as follows. Starting first with a fully accurate estimation model (zero error), we deliberately increased the error of estimation model up to 50%. We then measured the relative impact of the induced error by comparing the output of the result with the original solution of the controller when the error is zero. We define a sensitivity coefficient metric ($\psi$) that indicates the quality of the result achieved by the proposed controller (with regard to a particular performance metric $z$) when the estimation of the input parameter $x$ has an error of $\epsilon_x$. The experimental results confirm that even an error of 50% has a small negative impact in the solution quality (5.5% for $p$-99 latency and 3.9% for QoS violation incident rate).

4. Computational overhead: Figure 7 lists the computational overhead imposed by the controller to compute optimal solution for the allocation problem as the number of Flink applications increases. Particularly, the fraction of overhead time to find an optimal solution across a large platform with $n \in \{400, 1000, 4000\}$ running applications is below 0.8% of the controlling interval (as set to 1 second).

VI. RELATED WORK

To achieve SLA-awareness, a resource controller needs to adaptively fulfill the QoS requirements in distributed systems. Our solution follows the famous MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) paradigm that has been firstly introduced by IBM to design self-adaptive systems composed of highly configurable information processing components on distributed nodes [42]. By providing a mechanism to satisfy the system targets, the feedback control loop in our solution addresses an important challenge in the engineering system domains regarding the dynamically changing operating conditions.

The design of efficient scheduling schemes for real-time distributed stream processing received significant attention in recent years. Most of the studies addressed the limitation of RR (Round Robin) default scheduler of Apache Storm. For example, the offline algorithm proposed in [43] could successfully reduce the processing delays of streams when compared to the Storm default scheduler. However, such offline scheduling decisions require executing before an event is triggered in lieu of making schedule decisions during execution. Hence, its limitation is quickly revealed as it fails to adapt in run-time to varying traffic conditions.

The work in [44] presented an online mechanism to automatically explore the parallel level of a given topology based on the measured congestion status and the system throughput. It provided a solution regarding the stateful migration if rescheduling happens. The issue of stateful migration has
not been covered in our work, however, and we left it as a future work in this line. The online approach in T-storm [45] is concerned with the run-time traffic patterns. T-storm enables dynamic adjustment of schedule parameters to support running fewer worker nodes while speeding up the overall time for data processing. The evaluation showed that T-storm provides 84% and 27% speedup on lightly and heavily loaded topologies, respectively, while it achieves 30% less utilization of worker nodes compared with the default scheduler. However, T-storm does not support any mechanism to guarantee the QoS enforcements.

R-storm [46] implements a resource-aware scheduling in Storm by respecting to CPU and Memory constraints, network distance between components that communicate with each other, and the variety of resource types involved. Some node and task selection algorithms, which use the minimum Euclidean distance along the axis of CPU and network dimension (e.g., memory constraint), are never violated. Resource awareness makes R-storm achieving up to 47% throughput improvement and at least 69% improvement of CPU utilization compared to Storm default scheduler. None of the above-mentioned mechanisms gives an effective solution to handle changes in the arrival rate of tuples, workload pattern, and operating conditions particularly the slowdown caused by running multiple application together in a host, however.

Prior works proposed several multi-phase methodologies that enable controlling of resource allocation activities in either the data-center [47], [48] or the platform [49]–[51] level. In a fair strategic plan, a cluster management controller must ensure the end-user applications receive the minimum share of resources relative to others. This is a challenging problem because application’s demands for CPU and memory vary across different server nodes in the course of execution. To address such a challenge, a classical approach is to enforce proportional share of computing resources on each server node separately. For example, Authors in [52] developed a classification driven framework for parallelisation and elasticity in stream data processing. It leverages different types of programming model and memory architecture to support the extra demand in order to prevent parallelisation and elasticity in stream processing platforms; however, there is only a brief discussion on resource provisioning and task scheduling of streaming operators. In our work, we mainly focus on ensuring the quality of service for latency-critical workloads.

Authors in [53], [54] proposed a strategy to explicitly regulate the CPU frequency for the current load and also an algorithm to optimize energy consumption of server nodes through Dynamic Voltage and Frequency Scaling (DVFS) without affecting application SLA. Authors in [55] proposed a method based on exponential smoothing over the periodic observations of the incoming stream traffic rate to forecast the volume of future workloads and to help designers judge which placement strategy is best suited to a specific traffic condition. Differently, authors in [56], [57] proposed a class of mechanisms that boost performance (e.g., throughput) for data analytic workloads based on finding an strategic equilibrium in a multi-agent plan using game theory principals. In our work, however, we formulate the CPU cap and memory allocation problem in a shared server platform using model-predictive control theory principals while model each streaming operator as a \( G/G/N \) queue to help the controller forecast the impact of load fluctuations and manage data streaming applications during bursty periods.

VII. CONCLUSIONS

High-throughput and low-latency stream processing algorithms are of increasing importance due to their wide applicability in fields such as sensor-based monitoring applications in both military and industrial domains and large-scale distributed stream machine learning systems, which often require running continuous deep pipeline queries over real-time high-volume data. Measuring run-time system behavior can be of great practical importance to design a QoS-aware resource (e.g., CPU and memory) controller for a high-throughput low-latency stream processing engine. This paper described a distributed resource controller for Apache Flink engine to achieve such goals without the need for exchanging run-time states to a centralized system. The proposed solution enhances the QoS satisfaction level by 95%, while reducing the \( p\text{-}99 \) query response time up to 119% in the high-rate workload scenarios for applications with the highest QoS enforcement.

ACKNOWLEDGEMENT

Professors Zomaya and Tari would like to acknowledge the support of the Australian Research Council Linkage Industry scheme grant LP160100406. Professor Javid Taheri would like to acknowledge the support of the Knowledge Foundation of Sweden through the HITS Profile.

REFERENCES


**APPENDIX: BACKGROUND KNOWLEDGE**

1) Apache Flink: The emergence of Apache Hadoop was just the beginning of a long journey that makes it feasible to abstract the intrinsic complexity associated with scalable execution. Flink is built around the streaming first principle, which considers batch processing as a special case of streaming model [58]. Other data processing systems, such as Apache Spark, are designed based on a different design philosophy in which the input data remains unchanged during the course of execution; hence, they collect and treat data-streams as a set of mini batches. However, despite several intrinsic merits, the Hadoop ecosystem still suffers from several drawbacks including a low-level programming model interface and its lack of support for stateful and iterative computations. As a result, several alternative frameworks have been proposed to overcome such limitations at scale [59].

2) Architecture: Figures 8 and 9 depict high-level architectural overview and Flink’s main components. Flink applications are structured as directed graphs, also known as “JobGraph”. Each vertex of such graph represents a computational operation which can exchange data with other operations along the edge of JobGraph. Particularly, the two building blocks of an application are (i) intermediate result, representing a data-stream, and (ii) transformations which are stateful operators that take one or more streams as the input, and generates one
or more output streams. Each computation is attached to one or more data stream sources (e.g., IoT sensors, a Kafka topic or database transactions), and ends in one or more data stream sinks (e.g., file or database).

3) Deployment Plan: In the run-time, the logical computations as defined by distinct operators can be executed in a concurrent fashion across a multi-node Flink cluster. Such an execution model is in contract with traditional programming model (e.g., Map-Reduce) in which an application is firstly divided into individual phases and then each stage must be executed sequentially [12]. In Flink, operator subtasks are executed in the context of a distributed execution environment, and hence, must be designed with the capability to run independently from each other, for example, in different threads/containers across distributed nodes. The number of operator subtasks, referred to as the parallelism degree of an operator, is a configurable parameter that can be overwritten by the controller to improve the performance of application programs.

Different operators in a Flink application can have dissimilar degrees of parallelism while exchanging data among each others and following pre-defined forwarding patterns. The distributed engine in Flink adapts the execution plan to the cluster environment and can accordingly run distinctive configurable execution plans and/or data distribution deployments. Flink is compatible with a number of cluster management and storage solutions, such as Apache HDFS [60] and Apache Hadoop YARN [61]. Stream Builder and Common API components, as highlighted in Figures 8 and 9, are responsible to translate the represented schematic directed graphs of logical operations into generic data stream programs to be deployed on the Flink’s run-time environment. The query optimizer component finds an automatic optimization of data flow process, e.g., by finding the most preferment concrete filtering execution plan.

4) The Query Model: Different abstraction levels are provided by Flink to develop stream processing applications. Flink offers SQL-like query expressions which can be considered as the highest possible level of expressiveness. Using SQL APIs, a program semantic can be developed by a set of SQL-like queries which are executed over tables [12]. The table API in Flink is a declarative domain-specific language which is centered around dynamically changing streaming tables. It extends the relational table model, i.e., each streaming table is attached to a schema and can be declaratively manipulated using transformation operations such as select, project, join, group-by, and aggregate. The lowest level abstraction in Flink is provided using DataStream API and DataSet API which represent immutable collections of bounded or unbounded data sources. Using such APIs, application developer team can implement various kinds of transformations, such as filtering, updating state, defining windows, and aggregating, over different data sources, such as conventional files, Kafka topics, or in-memory data-stores.