Real-time Stream Data Processing at Scale

M. Reza HoseinyFarahabady∗, Ali Jannesari†, Wei Bao§, Zahir Tari‡, Albert Y. Zomaya¶∗,§,¶
∗,§,¶The University of Sydney, Center for Distributed & High Performance Computing,
School of Computer Science, NSW, Australia
†Department of Computer Science, Iowa State University, IA, USA
‡RMIT University, School of Science, Melbourne, VIC, Australia
∗,§,¶{reza.hoseiny, wei.bao, albert.zomaya}@sydney.edu.au, †jannesar@iastate.edu, ‡zahir.tari@rmit.edu.au

Abstract—A typical scenario in a stream data-flow processing engine is that users submit continuous queries in order to receive the computational result once a new stream of data arrives. The focus of the paper is to design a dynamic CPU cap controller for stream data-flow applications with real-time constraints, in which the result of computations must be available within a short time period, specified by the user, once a recent update in the input data occurs. It is common that the stream data-flow processing engine is deployed over a cluster of dedicated or virtualized server nodes, e.g., Cloud or Edge platform, to achieve a faster data processing. However, the attributes of incoming stream data-flow might fluctuate in an irregular way. To effectively cope with such unpredictable conditions, the underlying resource manager needs to be equipped with a dynamic resource provisioning mechanism to ensure the real-time requirements of different applications.

The proposed solution uses control theory principals to achieve a good utilization of computing resources and a reduced average response time. The proposed algorithm dynamically adjusts the required quality of service (QoS) in an environment when multiple stream & data-flow processing applications concurrently run with unknown and volatile workloads. Our study confirms that such a unpredictable demand can negatively degrade the system performance, mainly due to adverse interference in the utilization of shared resources. Unlike prior research studies which assumes a static or zero correlation among the performance variability among consolidated applications, we presume the prevalence of shared-resource interference among collocated applications as a key performance-limiting parameter and confront it in scenarios where several applications have different QoS requirements with unpredictable workload demands.

We design a low-overhead controller to achieve two natural optimization objectives of minimizing QoS violation amount and maximizing the average CPU utilization. The algorithm takes advantage of design principals in model predictive control theory for elastic allocation of CPU share. The experimental results confirm that there is a strong correlation in performance degradation among consolidation strategies and the system utilization for obtaining the capacity of shared resources in a non-cooperative manner. The results confirm that the proposed solution can reduce the average latency of delay-sensitive applications by 17% comparing to the results of a well established heuristic called Class-Based Weighted Fair Queuing (CWFQ). At the same time, the proposed solution can prevent the QoS violation incidents by 62%.

Index Terms—Stream Data-flow Processing, Dynamic CPU Cap Allocation, Shared Resource Interference

I. INTRODUCTION

Distributed data-flow and stream processing systems is a new class of in-memory computations which are designed with the main aim of handling an endless flow of data that generated by workloads which are commonly found in modern analytical applications. The application layer can be often represented as a composition of data source nodes, transformation operators, and a set of end-user components. Each processing element continuously receives the incoming data form the other components and generates some new outgoing streams after executing the requested processing logic. In most cases, the streaming analytical applications need to be performed in real time, i.e., each update in the output of the processing must be delivered to the user within a short period of time once an incoming update occurs.

A common characteristic among stream processing applications is that the data analysis needs to be run over the flow of data while it is in motion [1]. In such cases, a stream processing application is closely connected with the strict requirements on processing latency. Devising a dynamic CPU allocation strategy for such a platform is a recent active research area. The main aim is to provide an elastic mechanism to scale CPU (as the main computing resource) up or down whenever the rate of incoming data-tuples fluctuates. The occurrence of unexpected spikes in the incoming workload even further exacerbates the problem. Experimental results confirm that when a DDSP is operating at a significant traffic load, the total processing time of each data-tuple is determined by the waiting delays in the corresponding queues instead of the actual processing time. This suggests that models based on queuing theory can be used for designing an adaptive CPU cap controller to dynamically regulate the system parameters in response to the continuous feedback of the different component.

This paper proposes a novel adaptive architecture based on Model Predictive Control (MPC) design principles for elastic allocation of computing resources in real-time stream data processing platforms. The key feature of the proposed controller is to take QoS enforcements into account when making resource allocation decisions. The first contribution of this paper is that we use a distributed monitoring approach for monitoring the shared resource capacity (as sub-controlling module) embedded in each node. The sub-controllers in each node can exchange the local state information among themselves for making reconfiguration decisions in a distributed manner. The proposed algorithm identifies the QoS enforcement level in each machine by continuously monitoring the workload and the shared resource usage.
Another contribution of this paper is proposing a mechanism to dynamically evaluate and then try to mitigate the QoS violation incidents for every streaming application. We implement the propose controller as a novel pane-based partitioning technique for processing key-based window operators in SensorBee [2] which is a recent stream engine for low-latency processing of streaming data at the edge of the network and IoT devices in distributed and shared memory architectures. Experimental results show that the proposed approach outperforms some well-known scheduling heuristics by an average improvement of 24\% in reducing the average tuple latency of delay-sensitive applications, while enhancing the overall resource utilization by 26\%.

The rest of this paper is organized as follows. Section II concisely provides the reader with essential background knowledge and the system model targeted in this paper. Section III gives insights into the proposed controller. Section IV summarizes the experimental evaluation results, followed by a comparison with related work which is presented in Section V. Finally, Section VI draws some final conclusions.

II. BACKGROUND

There has recently been a considerable raise in adaption of scalable streaming data processing systems that enable the powerful concept of making decisions over near-real-time processing of large amounts of data generated from different sources [3], [4]. Particularly, the innovative concept is becoming more critical across different industries where the failure or success of a decision making process is determined by the response time to an external event [5]. One of the biggest challenges yet to answer is how to handle a large volume of streaming data in a short period of time [1]. A promising approach may be utilizing a distributed cluster consisting of tens of server nodes to process the streaming live data [6]–[8]. However, a number of challenges arise when building such a distributed high throughput platform. Devising a solution that can avoid single-node bottleneck in presence of highly fluctuating incoming traffic in a shared-resource environment is one of such difficulties.

The issue exacerbates when a service provider attempts to simultaneously satisfy incompatible objectives of end-users (such as fast execution time) and operator goals (such as high resource utilization and/or total monetary profit). To increase the resource utilization level, a service provider might be tempted to use a CPU cap strategy that packs as many application as possible in a single host. However, collocated applications may fiercely compete with each other to acquire shared resources such that the overall performance degrades significantly [9]–[11]. In addition, a QoS-oblivious dynamic CPU cap allocation algorithm might not be able to comply with the service-level objectives when there are unexpected spikes in the incoming traffic [12]. In such high input load scenarios, the processing time of each data item is mostly dominated by the total waiting time that each item spends in the relevant queue, instead of the actual processing time in the computing resources. Based on such fact, we exploit principles in dynamic controlling of queuing systems to design a QoS-aware CPU cap allocation for DDSP platforms that can effectively cope with the spikes in the incoming traffic.

Query model. Execution of user-defined queries/operations on streaming data tuples is different from the traditional “store-then-process” approaches that are conventionally used in the database management systems. In a DSP platform a query needs to be processed directly on incoming tuples in a real-time manner instead of storing them on non-volatile memory devices [13]. For example, in time-based sliding window model as a concrete implementation of “inbound query processing” concept, the result of a query is continuously evaluated/updated over the most recent data tuples [14], [15]. In sliding window model, the user can define two parameters of window length \( w \) and sliding length \( s \) to allow the query to be evaluated every \( s \) seconds over every data-tuple is received within the last \( w \) seconds, where \( w \) can be considered as the length of the sliding window [16]. So, at any given time instant, the portion of relevant data which is valid for answering queries (or gathering statistics) includes only set of tuples which arrives in the last \( w \) seconds. Using sliding-window model, both the memory space and the computational time required for the execution of each query can be reduced. We adapt this model in our project, due to its efficiency for processing data in fast arrival tuple rate cases. This model is widely used in most of the modern streaming data processing engines, such as Apache Storm [17] and SensorBee [2].

We also adopt pane-based aggregation model within each window processing. This model divides the overlapping windows (each identifies a finite subsequence of a stream) into smaller panes. Each pane can be considered as distinct subsequences of logically related data tuples belonging to a same window [18], [19]. By reducing the required buffer size and, at the same time, enhancing the corresponding computational cost, due to sharing of sub-aggregate outcomes and splitting of a window to multiple panes, improves the overall performance of a query executions over data streams [20].

Concurrency execution model. We adopt actor-based communication model to build a concurrent and distributed streaming data processing system. Each actor is considered as the building block primitive for creating a highly concurrent system consisting of possibly thousands independent computational units, each with its own local memory and processing logic, communicating via a message passing protocol. Each actor can determine a reaction to the received data by either modifying a local state to figure out the best way to respond to the subsequent incoming messages, e.g., by spawning more actors, or by exchanging new messages with others [21]. Contrary to the traditional error-prone global shared memory model, in which the parallel processing units share a global address space among each other, the actor model and the associated asynchronous non-blocking message passing communication pattern [22] can effectively alleviates us from having to deal with explicit locking mechanisms for thread/process management.

Shared resource interference. To enable a better scaling at
a lower cost and achieve both user and operator goals, a service provider is required to host hundreds of streaming processing applications using available computing resources. In many practical cases, however, such goals are incompatible with each other. The canonical example is the fast running time that is requested by the end-users versus the high level of resource utilization as the main objective of the service provider. To enhance the resource utilization, the service provider might be tempted to overuse the “consolidation” techniques to host multiple streaming applications into one host. However, each stream processing application could have its own resource utilization characteristics, QoS enforcement, and the sensitivity level to available computing resources. The contention among consolidated worker threads to obtain the CPU last level cache (LLC), memory bandwidth, and hard drive buffer (as the main three shared resources) is an impediment factor to design an effective consolidation solution. For example, the contention among different working threads to obtain shared cache capacity can cause an undesirable increase in the latency of other queries to process their own data which is supposed to be in the cache in the next CPU cycles. This fact has been addressed by several studies in other parallel distributed platforms like [9]–[11], [23].

III. THE DYNAMIC CPU-CAP CONTROLLER

This work considers CPU- and IO-bounded streaming data processing applications which share a (possibly virtualized) platform in a cluster of computing nodes (also know as a server farm). Our aim is to design a decentralized and self-adaptive strategy for controlling the computing resources to fulfill the performance requirements of multiple streaming applications when they share a computer system. We model a stream data processing application as a directed acyclic graph (DAG) in which the vertices represent the logic of some transformation operators that can receive, process and/or emit the data tuples across a set of particular edge to other vertices. The transformation operators can be a user-defined function and/or a set of sub-queries that together constitute a query of the end-user.

We adopt a model in which each user query can be considered as three distinguishable components of: (a) A source vertex, which its responsibility it to connect to outside data generation sources and receive a possibly unlimited stream of raw data; (b) transformation vertices which represent the logic of corresponding sub-queries. Each transformation operator receives the data tuples from the previous stage (upstream operators), and after applying the intended logic, pass the data tuple results to the subsequent (downstream) operators; and (c) a sink vertex, which continuously receives a set of output data tuples from the last-stage transformation operator, often by using a persistent storage device. To achieve a faster data processing platform, we exploit a parallelism model in which several instances of same operator can be created and executed in a distributed platform.

Figure 1 depicts the overall architecture of proposed approach to control the CPU cap of a stream processing platform when running multiple queries belonging to different data streams. The platform is structured as a pool of worker threads per each data stream in which a worker thread in stage j is responsible for handling the requests coming as the input data-tuples to that stage. Such a thread produces an output result to be further processed by corresponding worker threads in stage \( j + 1 \). At any given time \( \tau \), the proposed approach adjusts three parameters per each stream \( s \) as follows. (i) The number of worker threads in each stage \( j \) (ii) the amount of CPU share assigned to each worker thread, and (iii) the incoming tuple rate to the main buffer of subsequent stage. We also assign a shared buffer to store the unprocessed data-tuples belonging to that particular data stream. Particularly, the shared buffer can be implemented using a distributed queuing platform, such as Apache Kafka [24], to temporarily the set of store unprocessed data-tuples.

System model: We use Allen-Cunneen approximation of \( G/G/M \) queue [25], i.e., a queue that consists of \( M \) working threads with a general distribution of both arrival time and service time, to estimate an upper-bound of the average end-to-end delay time (sum of waiting time in the queue and the processing time) experienced by each data tuple. The A-C formula estimates the average waiting time of items (denoted as \( W_s \)) in any general \( G/G/M \) queue by computing the coefficient of variation for inter-arrival time and service time. Notably, while the A-C formula was developed using some computational-based estimation techniques, it gives a very good approximation to the average waiting time of customers in a \( G/G/M \) queue. As reported by Tanner in [26], the value obtained by the A-C formula were within 10% of their actual values in most practical scenarios.

Prediction module: We use the well-known auto-regressive integrated moving average (ARIMA) forecasting equation for estimating the future incoming rate to the first shared buffer of each streaming application.

Optimization module: To adjust the number of concurrently running instance of each operator at the next time-frame, shown by \( \eta^s \), the controller uses a simple formula as  
\( \eta^s_{t+1} = \frac{W_s}{T} \eta^s_t \),  
where \( W_s \) is the expected average waiting time of messages in the corresponding shared buffer. The initial value of \( T \) is set to \( T^0 = \frac{N}{T_{\text{max}}} \), where \( N \) is the largest possible number of transformation operators in the target streaming application, and \( T_{\text{max}} \) is the upper-bound
value, defined by the system administrator, that represents the acceptable total response time of the target streaming application. The controller progressively updates the value of $T^*_{\omega}$ based on the mean service time of each operator. Once $\eta^*$ is computed, the controller can calculate the amount of CPU to be allocated to each operator $\omega$, denoted by $C^*_{\omega}$. If available CPU capacity on each server node, denoted by $R$, is higher than the sum of requested CPU by all operators running in such a server, then the controller performs a cost-benefit analysis to identify the CPU share of each operator to be partially fulfilled in the next time-frame. The controller defines a contribution function that reflects the reward received by the service provider if it allocates a CPU share amount of $r_{\omega}$ to $\omega$. We define the contribution function as $C_{\omega}(r_{\omega}) = \mathcal{L}_{\omega} \times (r_{\omega} - C^*_{\omega})$ in which it complies with the famous discrete budget allocation problem [27]. It has a very fast solution based on the dynamic programming approach (details can be found in [27]). In this equation, $\mathcal{L}_{\omega}$ is a constant weight which represents the QoS class that the operator belongs to. At any given $\tau$, the optimization module use a dynamic programming approach to maximize the total contribution of the service provider formulated as follows:

$$\max_{\tau} \sum_{\omega} C_{\omega}(r_{\omega}). \tag{1}$$

To achieve the low latency requirements for stream applications with higher QoS level, the controller assigns more processing capacity to such operators. At the same time, it increases the flow rate of data-tuples from the shared queue of such streams into the corresponding sub-buffers. When a QoS violation incident is detected for such high-priority streams, the controller assigns more CPU share to those high-QoS streams which are suffering from QoS violation incidents. At the same time, the controller reduces the incoming rate of those streams with lower priority to reduce the negative effect of shared resource interference. The amount of such increment/reduction is calculated using the system model. Notably, the proposed controller does not assume any prior knowledge about the incoming rate of streaming data. It only monitors the run-time state of applications at each host to set the controlling actions, i.e., the incoming rate and CPU share for each operator.

IV. EXPERIMENTAL EVALUATION

We built a proof-of-concept prototype of the proposed streaming data processing platform and the CPU share controller on top of SensorBee, which is a lightweight engine designed specifically for low-latency processing of streaming data in edge computing and The Internet of Things (IoT) platforms. Such devices can generate large volumes of unstructured streaming data in such a way that determining the temporal characteristics of generated data is not possible to predict beforehand. The experiments have been performed in our local cluster consisting of four virtual nodes with a total 32 logical cores. Each VM is equipped with 8 GB of main memory and a 2.4 GHz Intel Xeon CPU with eight cores.

We performed a set of experiments using $N \in \{200, 250, 300, 350\}$ applications each performs the famous Kabsch algorithm [28] over synthetic data tuples. Such applications continuously compute the optimal rotation matrix that minimizes the root mean squared deviation by processing input data on-the-fly. This algorithm has extensive usage in graphics, chem-informatics, bio-informatics among others. More details of the analytical process and detailed implementation can be found in [29]. The generation rate of incoming data tuples for each application is chosen according to a Poisson distribution with a rate parameter of $\lambda = 150$ and $\lambda = 400$ tuples per second, namely non-burst and burst periods, respectively.

The QoS requirements of applications: We assume that there are exactly $Q$ different QoS classes in the SLA contract that the application owner can choose from ($Q$ is fixed to three in our experiments). A QoS class is a value pair, as $(q, \Delta T)$, where $r = \omega^*_{q,m}$ is the maximum delay that an application in class $q$ can tolerate. The $V_{q,\Delta T}$ denote an acceptable upper bound for the percentage of violation incidents experienced by applications in QoS class $q$ during an arbitrary interval of size $\Delta T$. Each application is randomly assigned to one of the three available different QoS classes. The parameters for the QoS classes are selected in a way that applications have vastly different QoS requirements, i.e., by being set to $V_{q=1,3} \in \{0.99,0.90,0.80\}$, and $\omega^*_{q=1,3} \in \{300,400,700\} \times 10^{-2}$ seconds, respectively.

Our main goal is to assess the adaptability of our controller to cope with sudden changes in the incoming traffic rates. We compared the proposed solution performance against the results obtained by two advanced heuristic algorithms, namely Weighted Round Robin (WRR) and Class-Based Weighted Fair Queuing (CWFQ), which is employed by most of commercial packages. WRR uses a round-robin policy to evenly balance the incoming traffic among the working threads. The number of workers per each QoS class in WRR is fixed to a value which is proportional to the priority of the QoS class. In CWFQ, the scheduler creates several classes of ready queues, each of which has its own buffer to run streaming applications based on their QoS class. Inside each class, the scheduler uses round-robin policy to evenly balance the incoming traffic among the working threads. An application can use more CPU shares if there is any unclaimed share by other classes.

Result Summary. Figure 2 depicts the average latency for
processing incoming streaming data tuples to compute Kabsch algorithm in different scenarios. The results are only depicted for applications belonging to the highest QoS class. The \( x \)-axis represents the number of applications used in each scenario (from \( N = 200 \) to \( 350 \)). Noticeably, the choice of CPU cap policy can affect the average processing time of streaming applications, from 14% (for \( N = 350 \) and \( \lambda = 400 \)) up to 24% (for \( N = 200 \) and \( \lambda = 150 \)). Further, the measurement of cluster-wide CPU utilization confirms that the proposed strategy can keep the utilization level of all machines 26% higher than what CBWFQ policy achieves.

Table I presents the 99th percentile latency as well as the average reduction in QoS violation incidents which are experienced by each streaming application belonging to different QoS classes when the incoming workload rate is higher than the available capacity of the underlying platform (i.e., \( N = 350 \) and \( \lambda = 400 \)). Because the available computing capacity is not enough to satisfy the whole demands, the controller uses a cost/benefit analysis, described by Eq. (III), to trade-off between the QoS violation rates and the performance level of applications in the highest QoS classes (\( QoS_{1,2} \)) to keep their latency close to the desirable target performance (such improvement is shown in the last column of Table I). The proposed solution can reduce the QoS violation incidents on average by 62% for streaming operators in the highest QoS classes. The main reason is that both WRR and CQWFQ policies assigns some portion of CPU resources to \( q_3 \) operators even if the monitored performance values for \( q_1 \) and \( q_2 \) are far from the desirable level.

**Computational Overhead.** As a result of using a dynamic programming approach to find an optimal solution for the allocation problem, the computational overhead of running the proposed controller is negligible. Particularly, the fraction of time for running the proposed solution to control a large-scale platform with 32 virtual servers and 1000 streaming applications is no more than 0.07% of the entire controlling time span.

### V. Related Work

Examples of stream processing applications can be mostly found in large commercial and industrial sectors, including financial markets, transportation, energy management systems, and health-care systems, to derive insight on critical business decisions. In such contexts, identifying the success or failure of a decision heavily relies on having efficient tools to process large volumes of data across a variety of data types in a real-time fashion. The emergence of numerous successful stream processing engines (including but not limited to StreamBase [30], IBM InfoSphere Streams [31], S4 [32], Apache Storm [17], STREAM [33], Borealis [34], and System S [35]) in both industry and academia indicates a strong future growth of the modern computational paradigm.

Apache Spark [36], originally developed at the University of California, Berkeley, is an open source platform which for performing large-scale data-intensive applications on commodity clusters. In Spark, the fundamental data structure is an immutable distributed collection of objects, namely Resilient Distributed Datasets (RDD). Spark’s core concepts is that of storing data items into RRDs which partitioned across several cluster nodes.

Beside processing latency, another important performance metric for streaming-native platforms is the maximum frame rate such that the shared buffer of input data does not over-flow, the so-called sustainable input rate. With regard to this metrics, Experimental results confirmed that Apache Storm outperforms the Apache Flink and Apache Spark frameworks due to its simpler logic to deliver data-items. Also, Storm topological structure is defined by the application developer, which improves the efficiency of the platform [42].

Applying techniques in model predictive controller (MPC) is a recent technology in this context, e.g., [43], [44], in which some models are built before the controller automatically allocates the right amount of resources to each application. One of the distinctive feature of the proposed solution in this paper with similar works is that we first estimates the degraded performance level of each streaming application by using a queue-based model and applying a prediction model for the future traffic rate to each application.

### VI. Conclusions

In this paper, we proposed a control mechanism to proactively alleviate the resource contention between collocated streaming processing applications in a shared and latency-sensitive environment. We modeled each inter-connected streaming operator as a \( G/G/k \) queue and employed Allen-Cunneen approximation to give an upper-bound of the buffering time experienced by every data tuple on average. Since there is no exact formula known for such a queue, A-C approximation provides a nearly accurate results under heavy traffic and is particularly suitable for performance prediction of stream processing applications. We also proposed using a module for predicting the incoming changes in the input traffic. This lays the foundation for a CPU cap controller that yields less QoS detriments over all available server nodes.

As a future work, we plan to investigate optimized strategies for migrating the data-flow system from the current configuration to the new one recommended that supports stateful iteration computations, such as the one proposed by [45].

<table>
<thead>
<tr>
<th>QoS Class</th>
<th>( CQWFQ ) latency (Sec. ( \times 10^{-2} ))</th>
<th>Ours latency (Sec. ( \times 10^{-2} ))</th>
<th>% Impr.</th>
<th>QoS violation incidents</th>
<th>% Impr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_1 )</td>
<td>317</td>
<td>294</td>
<td>7.3</td>
<td></td>
<td>71</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>346</td>
<td>299.8</td>
<td>13.4</td>
<td></td>
<td>54</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>413</td>
<td>465</td>
<td>-12</td>
<td></td>
<td>-29</td>
</tr>
</tbody>
</table>

**TABLE I**

*Left:* Comparison of the 99th percentile latency of streaming Kabsch algorithm as experienced by applications belonging to different QoS classes. *Right:* The average reduction in QoS violation incidents for the same scenarios. The rate factor is set to \( \lambda = 400 \).

...
ACKNOWLEDGMENT

We would like to acknowledge the support of the Australian Research Council Discovery-Industry Grant (DP190103710).

REFERENCES