Dynamic Control of CPU Cap Allocations in Stream Processing and Data-Flow Platforms

M. Reza HoseinyFarahabady*, Ali Jannesari†, Zahir Tari‡, Javid Taheri§, Albert Y. Zomaya¶
*,¶The University of Sydney, Center for Distributed & High Performance Computing
School of Computer Science, New South Wales, Australia
†Department of Computer Science, Iowa State University, IA, USA
‡RMIT University, School of Science, Melbourne, VIC, Australia
§Karlstad University, Sweden
Email: *reza.hoseiny@sydney.edu.au, †jannesar@iastate.edu, ‡zahir.tari@rmit.edu.au,
§javid.taheri@kau.se, ¶albert.zomaya@sydney.edu.au

***Abstract*—This paper focuses on Timely dataflow programming model for processing streams of data. We propose a technique to define CPU resource allocation (*i.e.*, CPU capping) with the goal to improve response time latency in such type of applications with different quality of service (QoS) level, as they are concurrently running in a shared multi-core computing system with unknown and volatile demand. The proposed solution predicts the expected performance of the underlying platform using an online approach based on queuing theory and adjusts the corrections required in CPU allocation to achieve the most optimized performance. The experimental results confirms that measured performance of the proposed model is highly accurate while it takes into account the percentiles on the QoS metrics. The theoretical model used for elastic allocation of CPU share in the target platform takes advantage of design principals in model predictive control theory and dynamic programming to solve an optimization problem. While the prediction module in the proposed algorithm tries to predict the temporal changes in the arrival rate of each data flow, the optimization module uses a system model to estimate the interference among collocated applications by continuously monitoring the available CPU utilization in individual nodes along with the number of outstanding messages in every intermediate buffer of all TDF applications. The optimization module eventually performs a cost-benefit analysis to mitigate the total amount of QoS violation incidents by assigning the limited CPU shares among collocated applications. The proposed algorithm is robust (*i.e.*, its worst-case output is guaranteed for arbitrarily volatile incoming demand coming from different data streams), and if the demand volatility is not large, the output is optimal, too. Its implementation is done using the TDF framework in Rust for distributed and shared memory architectures. The experimental results show that the proposed algorithm reduces the average and p99 latency of delay-sensitive applications by 21% and 31.8%, respectively, while can reduce the amount of QoS violation incidents by 98% on average.**

***Index Terms*—Dynamic CPU Resource Allocation, Timely Data-Flow Architecture, Scalable Data-Stream Processing**

I. Introduction

As a recent big data technology, stream processing is a computer programming paradigm that allows some applications to perform users queries over continuous data stream within a short time period as the corresponding events are occurring. There are multiple cases[1] in which the value of an organization comes from analyzing, understanding, and responding to its data while it degrades with time. In such environments, it is critical to have instant access to the analyzed data for taking timely and correct actions. While an organization might be tempted to use traditional databases and batch processing technologies, such as Map-Reduce programming model or Apache Hadoop, the most suitable existing tool for quickly responding to generated data and handle use cases which are requiring immediate actions needs to be found in streaming analytics engines. In a sophisticated platform, stream processing applications can use multiple computational units as a form of parallel/concurrent processing to be executed on each element in the data stream.

Timely dataflow is a general-purpose paradigm for implementing distributed streaming computations that may composed of several nested iterative computational blocks to be continuously run over incoming data [1]. The framework arose from a work carried out at Microsoft Research in 2013, where a group of researchers worked in creating a method to structure data processing computations across scalable and distributed platforms [1]. Unlike other streaming platform engines, such as MapReduce [2], Apache Flink [3], Apache Spark [4], Google MillWheel [5], Microsoft Sonora [6], and Apache Storm [7], the timely dataflow (TDF) paradigm can provide all aspects of expressive computations, iterative computations (with possibly many parallel tasks), and high performance at the same time [1]. Particularly, existing platforms only allow developers to write either restricted applications for parallel execution or, if they support expressive computations, it would result in an inefficient execution due to the synchronous mechanism to be manually regulated among parallel threads [8], [9]. On the contrary, the TDF paradigm provides some internal mechanisms for coordinating the fine-grained synchronous execution of parallel tasks. Such a mechanism is implemented using a logical time-stamp model attached to each data element. It enables the new paradigm to support developing applications that comprise of multiple *stateful*, *iterative* and *incremental* computations [1].

In this paper, we consider a class of dynamic CPU allocation problem as follows. We are given a set of TDF applications,

[1]such as stock markets, manufacturers, patient monitoring, and surveillance

each with a predefined maximum tolerable delay to accomplish its computation over incoming data. Each TDF application receives multiple streaming data-flows over time, each with a *rate* that is unknown to the service provider and can randomly change over time. A CPU allocation is an assignment of core capacity in the target system to each application by applying specific policies (e.g., prioritization, defining the CPU shares, CPU reservations, and limiting some settings if they are supported in the operating system layer). For example, using *Linux cgroup*, one can limit the access of a given process to a single CPU for 0.2 seconds in a 1 second window.

We developed a low-overhead Model Predictive Control (MPC) algorithm for the dynamic CPU allocation problem of disparate TDF applications, each with its own predefined performance level. The main feature of the proposed algorithm is considering QoS enforcement levels and continuously monitoring of available CPU share in each node when making CPU cap decisions. At each controlling interval, the algorithm estimates two important metrics of buffer length per each computational unit for every TDF applications and the average arrival rate of incoming flow to each buffer. The controller uses a light-weight mathematical model for such estimations as the actual measurement of these parameters is prohibitively expensive to be implemented in a real platform. When designing the CPU cap controller, both estimation values are considered to be inaccurate (model noise). The controller uses an optimization module to dynamically find a CPU cap decision that minimizes the amount of QoS violation incidents among all consolidated TDF applications over a finite-time horizon. The main reason that a service provider uses a consolidation strategy is to increase the utilization level of computing nodes. However, there is a consequential price to pay for using such consolidation strategies, particularly when applications fiercely compete with each other to access the CPU time of a server node [10], [11]. Such a competition can cause a significant performance degradation to be experienced by almost all consolidated applications in a shared environment [12], which is highly undesirable in practical situations, particularly when there are real-time processing requirements.

We evaluated the performance of the proposed solution against a number of empirical resource allocation strategies, including weighted round-robin, fixed priority scheduling, and Class-Based Weighted Fair Queueing (CFWFQ). Experiments have been conducted by running iterative generic-join algorithm, which is developed using TDF paradigm, under various workload intensities (with arrival rates of up to 5000 data items per second) with respect to two major performance metrics of response time and QoS violation rate. The experimental results confirm the effectiveness of proposed algorithm under heavy workload conditions. Particularly, the controller can decrease the average and p99 latency of applications in the highest QoS class up to 21% and 31.8% during the peak periods in comparison with the output of CBWFQ policy (which achieves the best result among others).

The rest of this paper is organized as follows. Section II concisely provides the reader with essential background needed to understand TDF programming model. 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

Designing scalable and fault-tolerant distributed systems for running parallel programs to process streaming data has been receiving increasing attention. This includes dealing with the upcoming issues in the processing of data-flow in near real-time fashion. We provide brief background information about the core concepts of a recent model on this domain, called *timely data-flow* (TDF) which is first introduced by Microsoft researches in 2013 [1], can be best described as a run-time paradigm for implementing and executing low-latency cyclic data-flow computational applications. Its promising feature lies in its capability to scale the same application up from a single thread on an ordinary computer to distributed execution across a cluster of hundred server nodes [13].

Unlike other distributed stream-based data processing platforms, scuh as MapReduce [2], Apache Spark [4], and Apache Storm [7] which require explicit synchronous and communication methods among processes and threads to provide efficient execution of parallel programs, the TDF provides developers with both high-level expressive computation and high performance in which they can implement sophisticated streaming computations with *iterative* control flow. The TDF kernel, with the compiler assistance, can deliver an executable with only a minimum synchronization overhead among processes and threads [13].

The new paradigm of *data-parallel dataflow* targets a certain class of streaming processing problems in which each computational operator (such as *filter*, *map*, *exchange*, *join*, and *reduce*) can fragment their input data elements between a number of independent working threads. The instructive rules inside each thread is elicited from a set of logic directives which are defined in the form of iconic closures (or methods) [13]. TDF platform can effectively assist developers to describe the entire computation/application as a set of instructions to continuously run as data-flow computations.

Such a high-level expressiveness can immensely simplify developing and testing applications which consist of several stages in the traditional parallel programming model, such as MPI[2], that requires developer to be responsible for handling all forms of communication patterns and/or explicit synchronization mechanisms among concurrent/parallel threads and processors. By capturing the instructions for the expressive computation, the TDF engine can seamlessly execute the same application, as a form of compiled executable file, with multiple working threads or processors on a single computer node, or by launching several threads/processors across a cluster with multiple nodes.

[2]Message Passing Interface

Another key aspect of TDF programming model is its ability to perform *iterative*, *stateful*, and *incremental* computations over a continues stream of data. To achieve this, the TDF paradigm introduces a new form of logical timestamp which is attached to each data element as a lightweight mechanism for supporting parallel, iterative, and incremental processing in a distributed environment. By observing such time stamps, each parallel worker knows exactly the number of outstanding data items that are still live – and might be running in another worker reside in a remote computer – and need to be delivered for further processing in a future time [1].

The DCG structure can consist of three system-provided vertices, namely *loop ingress*, *loop egress*, and *loop feedback* to construct a computation which is occurred within any nested loop context. Particularly, TDF offers a set of controlling vertices that specify the organization of the computation in a single loop context and to embrace the three advantages of prevalent models developed in the past for processing of large amount of streaming data. To create a TDF application, the developer team needs to express the entire computational units by precisely defining a directed cyclic graph (DCG) as the underlying dataflow graph which describes how the data flows from and to each operator. Correspondingly, every component within a nested loop needs to include at least one loop feedback vertex. The developers have to obey this structure in order to introduce any loop context if exists. The structural graph allows the TDF kernel to efficiently track the set of data records that might possibly flow throughout the computational components at any given time. Each computational unit may contain arbitrary code and methods to modify the state of input data messages. Such units can be used to exchange logically time-stamped messages along edges to/from other component using some user-defined stateful operators.

III. The Dynamic CPU-cap Controller

This section provides details of the proposed CPU cap controller in a TDF platform. The problem is becoming immensely challenging when multiple TDF applications with different QoS enforcement levels concurrently run in a distributed platform with shared computing resources. The QoS requirements are normally expressed as a set of performance specifications that specifies the quality of services[3] to be fulfilled. Nevertheless, translating the QoS requirements, *e.g.*, the average response time, to an accurate resource allocation decision, such as CPU cap, is not trivial. A static strategy that fully satisfies the peak demand of every application can excessively over-provision the assigned CPU cycles to few applications, hence causing a serious performance degradation for other co-located applications [14], [15]. Evidently, the uncertain demand for each computing resource, that might happen due to unexpected change in the incoming rate, exacerbates the hardness of such an assignment [16], [17].

We propose a feedback controller based on model predictive control theory principles to adjust the CPU share for every computational unit in a TDF platform which hosts several concurrently running TDF applications. The main aim is to develop a robust strategy that fulfills the QoS performance level enforced by each application when the underlying platform operates at a significant incoming traffic load. We develop a system model, based on practical results in queueing theory, for estimating the queuing delay of each local data buffer. The idea is to *dynamically* postpone the decision about CPU cap adjustment for each worker to the run-time, where the controller can estimate the following performance metrics: (1) the available CPU capacity in each machine, and (2) the QoS violation rate per TDF application.

To estimate the average end-to-end delay that the execution of every query/operation on an incoming streaming might last, we leverage the fact that measuring the number of non-processed messages in the buffer is a good approximation of the average queuing time of the associated computational unit, hence, its response time. The proposed solution predicts the future arrival rate of data-flow per application over a finite-time horizon, and then dynamically makes CPU cap decisions based on both the current and the predicted future states. We apply the design principals of the model predictive control theory to provide a robust performance despite the system modeling errors and inaccuracies in the prediction of incoming traffic rates.

A model predictive controller solves an open-loop optimal control problem to obtain a sequence of actions to be applied across the underlying system at each sampling time, denoted by $\kappa \in \{T_0, T_1, \cdots\}$. Such a calculation is continuously repeated at the next sampling time through rearranged horizons [18]. At each $T_k = T_{k-1} + \Delta T$, where ΔT is a fixed interval length, the controller measures a set of performance metrics, denoted by $\mathbf{y}_\tau$, to determine a vector of tracking errors by comparing each metric with a corresponding envisioned value (also known as *reference trajectory* or *set-point* and denoted by $\mathbf{r}_\tau$). Reference trajectory defines the desirable value for the target performance metrics along which the underlying system needs to follow after a disturbance occurs. The controller needs to retain the error values, computed as $\epsilon_\tau = \mathbf{r}_\tau - \mathbf{y}_\tau$, within an acceptable range. The controllable variable is the amount of CPU share assigned to each computational unit, denoted by $\mathcal{C}_{C_i,\tau}$, across the entire platform.

The controller adopts a simple technique to gradually diminish the value of ϵ_τ within the upcoming sampling epochs, *i.e.*, it is *not* necessary for the underlying system to be driven back to the set-point reference trajectory immediately. We adjust the CPU cap in such a way that the error value in the next τ' steps exponentially vanishes. This can be achieved by applying the controlling actions in several steps instead of enforcing the complete decision once into the system. Formally, using a formula like $\epsilon_{\tau+\tau'} = e^{-\tau' \Delta T / \mathcal{T}_{ref}} \epsilon_\tau$, where ΔT is the sampling interval, and $\mathcal{T}_{ref}$ is a metric called *response speed factor*, which defines a mechanism for adjusting the maximum number of steps that the system is allowed to fade the error away. In this formula, $\Delta T / \mathcal{T}_{ref}$ can mitigate any negative impact which is caused by the inaccuracy in either the predic-

[3] *e.g.*, end-to-end latency, throughput, and scalability

tion or in the developed system models [19]. For example, by setting this ratio to $1/3$, the controller forces the output vector ($\mathbf{y}$) to smoothly converge toward the reference trajectory in the upcoming three epochs without causing a large over- or under-shooting from the desirable output value.

The controller has an internal system model to predict the future behavior over a predefined prediction horizon. At any given time τ, the controller uses two measured performance metrics of (1) buffer length of each computational unit, denoted by $|B_{i,\tau}|$, and (2) the average arrival rate to such buffers. The controller also exempts the monitoring module the need for gathering arrival flow rates of data items to each buffer or the need to estimate the required workload amount of each computational unit (*i.e.*, the required CPU share), which is a restraint factor imposed by some previous works *e.g.*, [20]. Such monitoring activities are prohibitively expensive operations to be implemented in a real-time system.

Semantic of QoS assurance. We assume that there are exactly Q different QoS classes that the application owners can choose from. A QoS class is effectively a value pair, denoted by $\langle \omega^*_{q,m}, \mathcal{V}_{q,\Delta T} \rangle$, where $\mathbf{r} = \omega^*_{q,m}$ is the maximum delay that an application in class q can tolerate to collect the result of a query over some incoming data elements. Specifically, $\mathcal{V}_{q,\Delta T}$ reflects the acceptable upper bound for the percentage of violation experienced by each application in class q during an arbitrary interval of size ΔT. For example, in a scenario with three QoS classes ($|Q| = 3$) and the upper bound values of $\mathcal{V}_{q=1..3} \in \{0.99, 0.85, 0.65\}$, the overall delay for collecting the result of a query for applications in the first QoS class needs to not be taken longer than ω^*_1 for 99% of the incoming data flows during any arbitrary interval. Otherwise, it is considered as a QoS violation incident.

System model. To control the buffer length of each computational unit, we use a well-known formula in "queuing theory", called Allen and Cunneen approximation of $G/G/M$ queue [21], which assumes general arrivals and general service patterns for a queuing system. Such an approximation imposes a low computational overhead to predict the average waiting time of non-processed data elements ($\hat{W}_M$) in a $G/G/M$ queue. It can be formulated as follows:

$$\hat{W}_M = \frac{P_{cb,M}}{\mu M(1-\rho)}\left(\frac{C_S^2 + C_D^2}{2}\right), \tag{1}$$

where $C_D = \sigma_D/E_D$ is the coefficient of variation for inter-arrival time, $C_S = \sigma_S/E_S$ represents the coefficient of variation for service time, and the term $\frac{C_S^2+C_D^2}{2}$ is often referred as the stochastic variability of the queue. The term $P_{cb,M}$ reflects the probability that all workers in the entire queuing system are busy (*i.e.*, the waiting time of a just newly arrived data element is above zero) [21]. It is worth mentioning that, although the Allen-Cunneen formula was originally evolved using some computational-based estimation techniques without any formal proof, it often gives a highly-accurate approximation to the average waiting time of customers in a general $G/G/M$ queue. As shown by Tanner in [22], its obtained value is within 10% of the actual values in most scenarios.

Prediction module. This module provides an estimation for the incoming traffic rates for every computational unit which is directly connected to an external source. The result is used by optimization module for the final decision of resource sharing. Particularly, at any sampling interval of τ and for any data-flow of s, the prediction module estimates the next arrival rate values, denoted by $\lambda_{\tau+\kappa,s}$ of such data-flow, across the next $\kappa > 0$ steps. If the probability distribution of λ is known in advance, we can simply apply the well-established stochastic techniques over the recent observations to project the future values. Otherwise, an estimation model such as the time-series analysis [23], Kalman filter [24], or auto regressive integrated moving average (ARIMA) model [25]) can be used. We use ARIMA model for the scope of this study which is an effective tool to forecast a time series using its past observations.

Optimization module. The controller determines the possibility of satisfying the total CPU cap requests collected from all computational units by summing them up and comparing with the total available CPU capacity in the cluster. In a multi-class service, the controller needs to allocate the CPU cap shares among different QoS classes appropriate to the performance target as enforced by each class. In case of *CPU scarcity*, however, the optimization module performs a cost-benefit analysis to maximize the best interest of service provider (by allocating CPU share to applications with the highest priority). The reward function serves as the gain of service provider when satisfying the requested CPU share of applications belonging to a particular QoS class.

We model this allocation problem as a classic discrete budgeting problem which has a fast solution based on dynamic programming [26]. Such a model allows the controller to prioritize demands coming from applications with the highest QoS requirements aiming to meet the requested performance specifications in case of limited CPU share. In the classic discrete budgeting problem, it is assumed that a project manager needs to allocate a budget of size R to a series of projects, where each allocation can yield to a certain amount of profit for the firm. For our problem, we use the notation of $\mathcal{C}^*_{c_i,\tau}$ to indicate the desirable CPU cap demanded by a computational unit c_i at any given time τ, and also assume that the symbol R_τ denotes the whole available computing capacity in the cluster. Accordingly, we can define a *contribution* reward function, denoted by $\mathcal{C}_{c_i}(r_{c_i})$, to represent a reward received by the service provider if it assigns r_{c_i} amount of CPU to the computational unit c_i. One possibility to define a reward function is formulated as follows:

$$\mathcal{C}_{c_i}(r_{c_i}) = \mathcal{I}(q_{c_i}) \times (r_{c_i} - \mathcal{C}^*_{c_i}), \tag{2}$$

where q_{c_i} represents the QoS class that each c_i belongs to, and $\mathcal{I}(q_{c_i})$ is a constant weight representing the importance of each QoS class. A very common model for discrimination in a multi-class platform is the relative differentiated service model proposed by [27]. Based on this model, a simple policy

specifies that the relative importance of each QoS class must relate to the desired target performance, *i.e.*, $\mathcal{I}_{q_i} = \mathbf{r}_{q_i}/(\mathbf{r}_{q_1} + \cdots + \mathbf{r}_{q_Q})$, where Q denotes the number of QoS classes and $\mathbf{r}_{q_i}$ is the desired performance target of each application in class q_i. This metric determines how each QoS class is performing relative to other classes. In case of CPU scarcity, we need to maximize the predefined reward function as follows:

$$\max_{r} \sum_{c_i} \mathcal{C}_{c_i}(r_{c_i}), \tag{3}$$

subject to the clear constraints of $r_{c_i} \leq \mathcal{C}^*_{c_i}$ and $\sum r_{c_i} = R_\tau$. We use a method based on dynamic programming to find a fast solution for the aforementioned optimization problem. Assuming that $r_{c_i} > 0$ can only take *discrete* values, such as $\{10\% \cdots 100\%\}$ of a CPU core capacity, we first find the partial contribution of having exactly R_κ CPU capacity to be allocated to c_i's where $i \geq \kappa$. We denote such partial assignment by $V_\kappa(R_\kappa)$. By recursively solving the following Bellman's equation, we can exactly find the optimal value for Equation (3).

$$V_\kappa(R_\kappa) = \max_{0 \leq r_\kappa \leq R_\kappa} \left(\mathcal{C}_{c_\kappa}(r_{c_\kappa}) + V_{\kappa+1}(R_\kappa - r_{\kappa_k}) \right), \tag{4}$$

where the initial step is $V_n(\mathcal{R}) = \max_{0 \leq r_{c_n} \leq \mathcal{R}} \mathcal{C}_{c_n}(r_{c_n})$, $\forall 0 \leq \mathcal{R} \leq R_\tau$, and n denotes the total number of computational units (see [26] for a formal proof).

Micro-architecture level interference. While workload consolidation remains as one the best solution to cope with the problem of low resource utilization in a large-scale data-center, the contention issue among consolidated workloads to obtain CPU cache or memory bandwidth remains a major impediment for devising an effective consolidation method [28]. Some authors suggested that the micro-architecture level interference among collocated applications can be detected by performing some forms of *off-line profiling* techniques over the target applications [32]. We also realize that applying such off-line profiling techniques exhibits at least two major shortcomings, as also reported by previous studies such as [33]. First, it is not always practical to perform such an off-line profiling over the submitted applications in real scenarios, particularly if they are constantly submitted by the end-users. Second, the traits of a single application can significantly change over the execution time, which makes the result obtained by performing an off-line profiling useless for live consolidation decision.

We try a different strategy, based on a method initially proposed by [34], for quantifying the slowdown level due to the interference in the micro-architecture layer. Based on this strategy, the impact of workloads' contention on both last level cache and memory bandwidth can be identified when there is any abnormal rise in the *memory bandwidth utilization*. The memory bandwidth utilization level can be calculated by analyzing two standard hardware events of *UNC_NORMAL_READS*, as an indicator of memory reads, and *UNC_WRITES*, as an indicator of memory writes. The CPU cap controller avoid to assign a new application to a host which currently experiences a level of memory bandwidth utilization higher than a threshold value (set to 80% in our experiments).

IV. Experimental Evaluation

We performed a series of experiments using synthetic large-scale data flows over an implementation of timely data-flow framework written in Rust [13] to validate the proposed CPU cap algorithm. The main goal is to examine the adaptive behavior of the autonomic controller under sudden changes in the incoming traffic rate to each data flow. Its effectiveness is carried out pertaining to: (i) the average and 99^{th} percentile of processing latency as experienced by each data flow, (ii) the amount of QoS violations occurring in the entire system, and (iii) the robustness of the proposed controller against errors in the system model, *i.e.*, Eq. (1), or in the prediction module.

We compared the performance results of the proposed solution against the ones obtained by three empirical heuristics implemented in most of commercial packages, namely Weighted Round-Robin (WRR), Fixed Priority Scheduling (FPS), and Class-Based Weighted Fair Queueing (CFWFQ). The WRR heuristic uses a round-robin policy to evenly balance the incoming traffic among the working threads or processes. According to WRR, the number of workers per each QoS class is fixed to a value which is proportional to the priority and the total number of QoS classes in the platform.

The scheduler of FPS assigns a fixed rank to each application and then sorts them in a ready queue in order of their priorities. So, applications with lower-priority can use CPU share only after all higher-priority applications have already finished their operations. In CBWFQ, the scheduler creates several classes of ready queues, each of which has its own buffer to host applications based on the QoS class to which the application belongs. Each ready queue receives a minimum reserved CPU share. Further, an application can use more CPU shares if there is any unclaimed share by other classes.

The reported experimental results have been performed in a local cluster consisting of sixteen server nodes with a total 32 logical cores. Each machine is equipped with 8 GB of main memory and four 2.40-GHz CPU cores. We use an approach firstly proposed by authors in [17] to translate the output of optimization solver, *i.e.*, Eq. (3), to the amount of CPU core utilization in each server node. We created $m \in \{150, 250, 350, 450\}$ TDF applications, each consists of three computational units to perform iterative generic-join algorithm. We assign each application to one of the three possible QoS classes. The QoS parameter for each class is set to $\mathcal{V} \in \{0.99, 0.85, 0.75\}$. The generation rate of incoming data for each application is chosen according to a Poisson distribution with an average rate parameter fixed to 5,000 data items per second. We varied the number of entangled relations in each scenario, *i.e.*, $\varrho = |\Phi|$, from three to five (which are ordered in Figure 1 from left to right). We also defined a *density* factor, denoted by κ, which indicates the ratio of total number of records to the total number of distinct values each relation. In other words, if each relation is considered as a graph $G = (V, E)$, the density factor can be seen as $\kappa = \frac{|E|}{|V|}$.

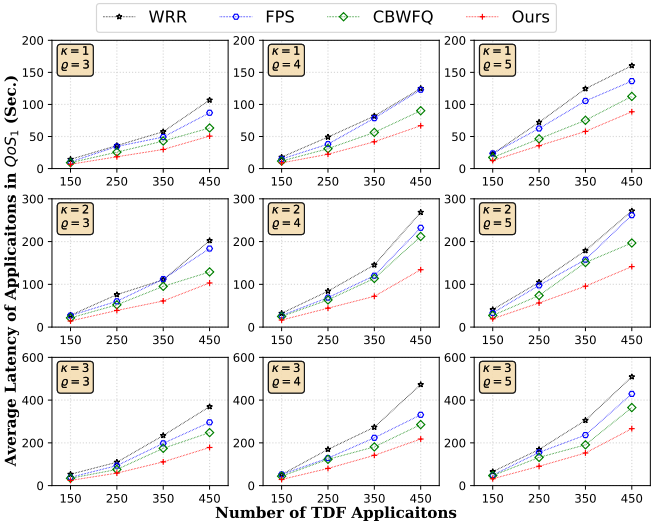

Fig. 1. Improvement in the average latency of applications belonging to the highest QoS class when the proposed solution for CPU cap management is employed. The scenarios are differentiated with respect to the density factor and the degree of entanglement among relations. The x and the y axes represent the total number of TDF applications and the mean response-time for finishing a join operation over input data, respectively.

We increased the density factor from one to three to create different scenarios (which are ordered from top to bottom in Figure 1). The larger value of ϱ, the higher number of iteration to be performed by generic-join algorithm. Likewise, the higher value of κ, the larger amount of data records needs to be processed by the algorithm.

Result Summary. Figure 1 depicts the average delay of generic-join algorithm implemented in TDF platform in multiple scenarios. The algorithm performs over data-flows with different parameters of κ and ϱ with the total input size of 50 million raw data items. The results are only shown for applications belonging to the highest QoS class. The horizontal axis represents the total number of streaming applications used in each scenario (increasing from $N = 150$ to 450). We observed that the choice of CPU allocation policy can significantly affect the average processing time of join algorithm, from 21% (for $\kappa = 1$ and $\varrho = 3$) up to 140% (for $\kappa = 3$ and $\varrho = 5$). Notably, such discrepancy in performance improvement is more significant in scenarios whit more complex iterative operations and more voluminous data records, *i.e.*, higher κ and ϱ. In addition, the measurement of cluster-wide utilization in all CPU cores confirms that the proposed control strategy can keep the utilization level of CPU cores between 55% and 90% when the workload demand is high. While the average CPU utilization when applying CBWFQ policy, which shows the best performance among others, is up to 73%. Overall, the proposed controller enhances the average utilization of all CPU cores by 28.3% (max 46.9%) in comparison with the best outcome of other heuristics.

Table I shows the 99^{th} percentile latency and the average reduction in QoS violation incidents as experienced by

TABLE I

THE 99^{th} PERCENTILE LATENCY OF JOIN APPLICATIONS AND THE AVERAGE REDUCTION IN QOS VIOLATION INCIDENTS AS EXPERIENCED BY TDF APPLICATIONS IN DIFFERENT CLASSES. THE PROPOSED SOLUTION IS COMPARED WITH THE RESULT OF BEST OUTCOME AMONG OTHER HEURISTICS ($\varrho = 5$ AND $\kappa = 3$).

QoS Class	$p99$ latency			QoS violation incidents
	Best	Ours	Impr.	Impr.
q_1	438.1	298.4	31.8%	98%
q_2	502.6	360.8	28.2%	23%
q_3	532.9	586.3	-10.0%	-38%

applications in different QoS classes under heavy workload (*i.e.*, when $\kappa = 3$ and $\varrho = 5$). During this experiment, the available CPU capacity in the entire platform was not enough to fully satisfy the entire demands requested by all applications. Therefore, the controller uses the cost/benefit analysis to trade-off among the QoS violation rates to prioritize the performance level of applications belonging to the highest QoS class. As a result, the controller assigns more CPU share to applications in QoS_1 and QoS_2 to keep their latency close to the target performance metric. The improvement in reducing the number of violation incidents per QoS class is listed in the last column of Table I.

Computational time overheads. The execution time to find an optimal solution using the proposed dynamic programming method remains below 0.12 milliseconds using a C implementation for a scenario with 450 TDF applications across a cluster with eight machines and with the sampling period of 1 second. As the running time grows linearly with regard to the number of applications and the total number of nodes, it can be considered as a practical solution in a cluster with hundreds of nodes running thousands of applications.

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 implemented two mechanisms to increase the robustness of the solution in spite of such noises. First, the controller chooses a *response speed factor* strictly greater than one, *i.e.*, $\Delta T/\mathcal{T}_{ref} > 1$ (3 in our experiments), which enabled it to *gradually* apply controlling decisions through multiple steps. Secondly, by using a low-pass filter, the controller reduced the uncertain noise associated with the prediction model.

Table II represents the sensitivity of the result as there are injected inaccuracies in the estimation tool. 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 90%. 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 (ψ) 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

TABLE II
THE SENSITIVITY COEFFICIENT (%) FOR LATENCY AND QOS VIOLATION INCIDENTS AS THE PREDICTION ERROR VARIES FROM 10% TO 90%.

Estimation Error [%]	Change in Latency [%]	Change in QoS Violation Incidents [%]
10	0.8	0.6
50	1.7	1.4
90	4.1	3.5

$\epsilon_{\mathbf{x}}$. Precisely, $\psi_{\epsilon,\mathbf{z}}$= $|\mathbf{z}(\mathbf{x}) - \mathbf{z}(\mathbf{x} \pm \epsilon)|/\ |\mathbf{z}(\mathbf{x})|$. The experimental results confirm that even an error of 90% has a light negative influence in the solution quality (4.1% for the response time and 3.5% for the QoS violation rate).

Skewness. Workload skewness can be defined as the uneven distribution of data/work across parallel/distributed workers. Such an imbalance can potentially lead to lingering processing and under-utilization of the computing resources [35]. We measured the size of intermediate buffers on every computational units during its course of execution under various volume of incoming raw data to observe the workload skewness caused by applying different strategies. We define the imbalance factor as the ratio of maximal buffer size on task instances among all TDF applications to the average buffer size of each individual application (*i.e.*, $\gamma = \frac{\max L_{c_i}}{\bar{L}}$), which L_c denotes the buffer length of a given computational unit. The results of load imbalance factor achieved by different strategies are shown in Figure 2.

The experimental result confirms that workload imbalance is highly relevant to the operated CPU cap strategy, and such an imbalance problem is exacerbated at scale as the size of incoming data grows. Unlike other heuristics which strive to distribute the number of tuples among workers as evenly as possible without considering the run-time conditions and accumulated loads in each buffer, our solution exploits the skew in the intermediate buffers by dynamically lowering/increasing the assigned CPU cap to the corresponding computational unit. Experimental results also confirm the effectiveness of our dynamic solution which does not need a global coordination among working threads. Figure 2 depicts the load imbalance factor as a percentage achieved by four algorithms in different scenarios as the total size of incoming data to be processed using TDF join applications grows. The proposed controller can mitigate the workload imbalance by up to 49% for TDF applications in the highest priority class in comparison with the best outcome of other heuristics, which is achieved by CBWFQ.

V. RELATED WORK

Distributed data-flow processing has been effectively employed in the field of big data mining where algorithms show iterative nature. Naiad [1] has been designed as a distributed system for running parallel and iterative operations over both batch and streaming data-flows. In Naiad, each message has a logical time-stamps that allows the underlying system figuring out the right order and the associated priority of each message.

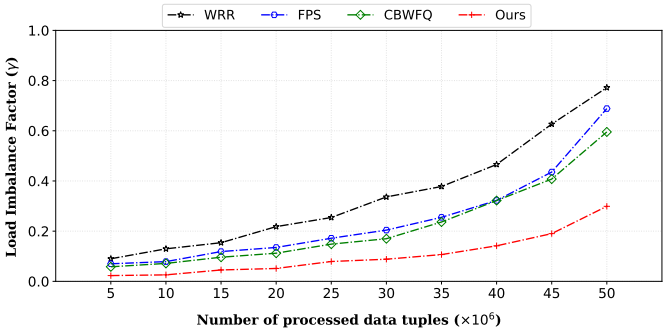

Fig. 2. Average load imbalance factor of stream generic-join algorithm after processing a stream of $m \in \{5 \cdots 50\}$ million data tuples on a data-set with fixed values for $N = 450$, $\varrho = 5$ and $\kappa = 3$. The figure confirms the impact of data size and dynamic resource allocation policy on load imbalance in a TDF platform. The x and the y axes represent the total number of data tuples processed by each TDF application and the average load imbalance factor achieved by different resource allocation policies after finishing the join operation over input data, respectively.

However, the thread-level elasticity in Naiad system is not fully supported.

Devising a dynamic CPU allocation strategy for a distributed platform is a popular research area. The main aim is to design an elastic mechanism to scale CPU (as the main computing resource) up or down whenever the rate of incoming workload fluctuates [15]. This problem has been well investigated in traditional distributed platforms [33], [36]–[41], where different methods are used (e.g threshold-based rules for CPU utilization) to decide when and by how far add and remove a computing resource. Most of such strategies manage the computing resources based on OS level metrics, such as per core utilization, I/O capacities, and energy usage of resources while ignoring the negative performance caused by interference at the shared resources.

However, a careful study by [34] confirmed that any resource management scheme that is unaware about the interference of shared resources is entirely a failure. Such a mechanism is necessary to avoid the performance degradation problem caused by consolidation decision among collocated workloads. Using model predictive controller (MPC) is a recent technology in the domain of distributed and parallel systems, like the work proposed in [36], [37], [42], [43], where a relatively complex relationship between the application performance and each resource allocation decision is built. The proposed MPC controller automatically allocates the right amount of resources to each application based on some prediction and system models. As a novel scheme in this context, our proposed solution estimates the degraded performance level of each applications by using a queue-based model that determines the number of lagged events.

VI. CONCLUSIONS

In this paper, we proposed a low-overhead feedback-driven resource allocation mechanism that dynamically adjusts the CPU cap share for every co-running TDF applications. It comprises of a model predictive controller that employs an

optimization module to fulfill application's quality of service enforcement. The effectiveness of proposed solution has demonstrated an average improvement of average and $p99$ latency of iterative join applications in the highest QoS class by 21% and 31.8%, respectively in comparison with the best outcome of three other heuristics (*e.g.*, Weighted Round-Robin (WRR), Fixed Priority Scheduling (FPS), and Class-Based Weighted Fair Queueing (CFWFQ)). It also reduces the QoS violation incidents on average by 98% for applications in the highest QoS class comparing to the result of CBWFQ. As a future work, we will investigate the theoretical properties of the robustness of the proposed controller to better understand the optimality of the approach under general assumptions about workload fluctuations.

ACKNOWLEDGMENT

We would like to acknowledge the support of the Australian Research Council Linkage-Industry Grant (LP150101213). Also, we would like to extend our thanks to ATMC Pty Ltd for their support of this work as the industry partner for LP1501013.

REFERENCES

[1] D. Murray *et al.*, "Naiad: a timely dataflow system," in *Sym. on Operating Systems Principles*. ACM, 2013, pp. 439–455.
[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of ACM*, vol. 51 (1), pp. 107–113, 2008.
[3] "Introduction to Apache Flink," flink.apache.org, accessed: Jun. 2019.
[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.
[5] T. Akidau and A. Balikov, "Millwheel: fault-tolerant stream processing at internet scale," *Proc. of VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.
[6] F. Yang, Z. Qian, X. Chen, I. Beschastnikh *et al.*, "Sonora: A platform for continuous mobile-cloud computing," *Microsoft Research Asia, Tech. Rep*, 2012.
[7] Apache Software Foundation, "Storm, an open source distributed real-time computation system," 2016. [Online]. Available: http://storm.apache.org/
[8] P.-N. Clauss and J. Gustedt, "Iterative computations with ordered read–write locks," *J. of Parallel & Distributed Computing*, vol. 70, no. 5, pp. 496–504, 2010.
[9] S. Dudoladov, C. Xu *et al.*, "Optimistic recovery for iterative dataflows in action," in *SIGMOD Intl. Conf. on Management of Data*. ACM, 2015, pp. 1439–1443.
[10] M. Hirzel *et al.*, "A catalog of stream processing optimizations," *Computing Surveys*, vol. 46, no. 4, p. 46, 2014.
[11] M. Stonebraker *et al.*, "The 8 requirements of real-time stream processing," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.
[12] P. Tembey *et al.*, "Merlin: Application-and platform-aware resource allocation in consolidated servers," in *Symp. on Cloud Computing*. ACM, 2014, pp. 1–14.
[13] F. McSherry, "A modular implementation of timely dataflow in Rust," https://timelydataflow.github.io/timely-dataflow/, 2017-11-1.
[14] Y. K. Kim *et al.*, "Decentralized admission control for high-throughput key-value data stores," in *Cluster, Cloud & Grid Computing*. IEEE, 2018, pp. 503–512.
[15] Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, A. Y. Zomaya, and R. Jurdak, "Dynamic control of cpu usage in a lambda platform," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 234–244.
[16] C. Delimitrou and C. Kozyrakis, "Resource-efficient provisioning in shared clouds," in *Operating Systems Review*, vol. 50, no. 2. ACM, 2016, pp. 473–488.
[17] O. Adam *et al.*, "Ctrlcloud: Performance-aware adaptive control for shared resources in clouds," in *Cluster, Cloud & Grid Computing*, 2017, pp. 110–119.
[18] J. Lee and H.-J. Chang, "Analysis of explicit model predictive control for path-following control," *PloS one*, vol. 13, no. 3, p. e0194110, 2018.
[19] J. B. Rawlings and D. Q. Mayne, *Model predictive control: Theory and design*. Nob Hill Pub., 2009.
[20] M. R. Hoseinyfarahabady, N. Farhangsadr *et al.*, "Elastic cpu cap mechanism for timely dataflow applications," in *Intl. Conf. on Computational Science*. Springer, 2018, pp. 554–568.
[21] A. O. Allen, *Probability, statistics, and queueing theory*. Academic Press, 2014.
[22] M. Tanner, *Practical queueing analysis*. McGraw-Hill, 1995.
[23] N. R. Herbst, N. Huber *et al.*, "Self-adaptive workload classification & forecasting for proactive resource provisioning," *Concurrency & computation: practice & experience*, vol. 26 (12), pp. 2053–2078, 2014.
[24] K. Rudolph, "A new approach to linear filtering and prediction problems," *J. of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.
[25] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.
[26] W. B. Powell, *Approximate Dynamic Programming: Solving the curses of dimensionality*. John Wiley & Sons, 2007, vol. 703.
[27] C. Dovrolis, D. Stiliadis, and P. Ramanathan, "Proportional differentiated services: Delay differentiation and packet scheduling," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4, pp. 109–120, 1999.
[28] J. Mars *et al.*, "Bubble-up: Increasing utilization in modern warehouse scale computers," in *Symp. on Microarchitecture*. ACM, 2011, pp. 248–259.
[29] H. Yang, A. Breslow *et al.*, "Precise online QoS management for increased utilization in warehouse scale computers," in *SIGARCH Computer Arch. News*, vol. 41 (3). ACM, 2013, pp. 607–618.
[30] R. Moraveji, *et al.*, "Data-intensive workload consolidation for the hdfs," in *Grid Computing*. IEEE, 2012, pp. 95–103.
[31] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "A unified mechanism to address both cache pollution and thrashing," in *Parallel architectures & compilation techniques*. ACM, 2012, pp. 355–366.
[32] K. Ye, Z. Wu, C. Wang *et al.*, "Profiling-based workload consolidation and migration in virtualized data centers," *IEEE Transactions on Parallel & Distributed Systems*, vol. 26, no. 3, pp. 878–890, 2015.
[33] M. R. HoseinyFarahabady, A. Y. Zomaya, and Z. Tari, "A model predictive controller for managing qos enforcements and microarchitecture-level interferences in a lambda platform," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 7, pp. 1442–1455, July 2018.
[34] H. Wang, C. Isci, L. Subramanian, J. Choi, D. Qian, and O. Mutlu, "A-DRM: Architecture-aware distributed resource management of virtualized clusters," *SIGPLAN Notices*, vol. 50, no. 7, pp. 93–106, 2015.
[35] J. Fang, R. Zhang, T. Z. Fu *et al.*, "Parallel stream processing against workload skewness and variance," in *High-Performance Parallel & Distributed Computing*, ser. HPDC '17. NY: ACM, 2017, pp. 15–26.
[36] K. Li, C. Liu, and K. Li, "An approximation algorithm for scheduling simple linear deteriorating jobs," *Theoretical Computer Science*, vol. 543, pp. 46–51, 2014.
[37] X. Huang, G. Xue, R. Yu, and S. Leng, "Joint scheduling and beamforming coordination in cloud radio access networks with qos guarantees," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 7, pp. 5449–5460, 2016.
[38] B. Jennings and R. Stadler, "Resource management in clouds," *J. of Network & Systems Management*, vol. 23, no. 3, pp. 567–619, 2015.
[39] C. Qiu, H. Shen, and L. Chen, "Probabilistic demand allocation for cloud brokerage," in *Computer Communications*. IEEE, 2016, pp. 1–9.
[40] Z.-H. Zhan, X.-F. Liu, Y.-J. Gong *et al.*, "Cloud computing resource scheduling and a survey of its evolutionary approaches," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, p. 63, 2015.
[41] M. R. Hoseiny Farahabady, H. R. Dehghani Samani, Y. Wang, A. Y. Zomaya, and Z. Tari, "A qos-aware controller for apache storm," in *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, Oct 2016, pp. 334–342.
[42] T. De Matteis and G. Mencagli, "Keep calm & react with foresight: Strategies for low-latency & energy-eff. elastic data stream proc." in *Principles & Practice of Parallel Programming*, 2016, p. 13.
[43] G. Mencagli, "Adaptive model predictive control of autonomic distributed parallel computations with variable horizons and switching costs," *Concurrency & Computation: Practice & Experience*, vol. 28, no. 7, pp. 2187–2212, 2016.