Elastic CPU Cap Mechanism for Timely Dataflow Applications

M. Reza Hoseinyfarahabady[1(✉)], Nazanin Farhangsadr[1], Albert Y. Zomaya[1], Zahir Tari[2], and Samee U. Khan[3]

[1] School of IT, Center for Distributed and High Performance Computing, The University of Sydney, Sydney, Australia
{reza.hoseiny,albert.zomaya}@sydney.edu.au
[2] School of Science, RMIT University, Melbourne, Australia
zahir.tari@rmit.edu.au
[3] Department of Electrical and Computer Engineering, North Dakota State University, Fargo, USA
samee.khan@ndsu.edu

Abstract. Sudden surges in the incoming workload can cause adverse consequences on the run-time performance of data-flow applications. Our work addresses the problem of limiting CPU associated with the elastic scaling of timely data-flow (TDF) applications running in a shared computing environment while each application can possess a different quality of service (QoS) requirement. The key argument here is that an unwise consolidation decision to dynamically scale up/out the computing resources for responding to unexpected workload changes can degrade the performance of some (if not all) collocated applications due to their fierce competition getting the shared resources (such as the last level cache). The proposed solution uses a queue-based model to predict the performance degradation of running data-flow applications together. The problem of CPU cap adjustment is addressed as an optimization problem, where the aim is to reduce the quality of service violation incidents among applications while raising the CPU utilization level of server nodes as well as preventing the formation of bottlenecks due to the fierce competition among collocated applications. The controller uses and efficient dynamic method to find a solution at each round of the controlling epoch. The performance evaluation is carried out by comparing the proposed controller against an enhanced QoS-aware version of round robin strategy which is deployed in many commercial packages. Experimental results confirmed that the proposed solution improves QoS satisfaction by near to 148% on average while it can reduce the latency of processing data records for applications in the highest QoS classes by near to 19% during workload surges.

Keywords: Shared resource interference
Distributed stream processing
Scheduling and resource allocation algorithms

Y. Shi et al. (Eds.): ICCS 2018, LNCS 10860, pp. 554–568, 2018.
https://doi.org/10.1007/978-3-319-93698-7_42

1 Introduction

Timely data-flow is a recent powerful general-purpose low-level abstraction layer to be used in developing scientific/enterprise programs that consist of large-scale *iterative* computations over batch- or streaming- based data records [22]. The growing demand for fast analysis over a high volume of unstructured data records [3,16] has led to the development of several *acyclic* batch processing or streaming data processing such as MapReduce [11], CStream [26], and Microsoft Sonora [31]. However, an important feature of emerging applications in several domains, such as deep learning algorithms, is their need to *iteratively* execute certain modules with many parallel tasks over a large amount of data elements (either in batch or in streaming mode) until a termination condition is matched [13].

The goal of the timely data-flow model is to bring together all *three key advantages* of the previous computational models, namely (i) batch computational model, (ii) the streaming computational model, and (iii) the graph computational model, into a common paradigm, while retaining the performance of each system. To this end, the timely data-flow model supports both *stateful iterative* and *incremental computations* to coordinate the fine-grained synchronous and asynchronous execution of parallel tasks. The model involves a directed *cyclic* graph where its vertex set represents the computational tasks, each can send and receive logically timestamped *stateful* data elements along the directed edges of the graph. Nevertheless, the new model satisfies the three main requirements: (i) low-latency, (ii) synchronous/asynchronous iteration, and (iii) strict consistency of the intermediate sub-computational results, alongside each other [12].

On the other hand, handling the fast processing requirement over a large volume of data in a scalable manner, taking advantage of a server farm of tens or even hundreds of server nodes seems inevitable [14]. Two important design objectives of such large-scale systems are to employ parallelism techniques to attain a high scalable solution by avoiding single-node bottlenecks while the hardware resource usage needs to be utilized in a cost effective way. To enable better scaling at a lower cost, a service provider (SP) of such frameworks often chooses hosting tens of thousands of users applications on the available computing resources at the same time. While the main objective of the service provider is to maximize her revenue, an end-user (*e.g.*, an application owner) may demand fast execution time. However, Satisfying such incompatible objectives can disappointingly lead to under/over-utilization of precious cycles of computing resources in many practical cases.

Another critical requirement for a data-flow processing system is fulfilling the **quality of service** (QoS) requested by the application owners as specified in the **service level agreement** (SLA). While a dynamic resource allocation strategy can provision a new server once the capacity of the available resources is not enough to cope with the incoming load, there exists several practical scenarios in which it is almost impossible to avoid QoS violations for executing all applications over the course of their execution, particularly if there are unexpected

spikes in the incoming traffic. In case of such scenarios, a QoS-aware resource manager needs to carefully comply with the service-level objectives while maximizing the overall system performance.

This paper proposes a low-overhead feedback controller for elastic adjustment of CPU resources in a timely data-flow platform running on a shared distributed environment. The key features of our solution is taking QoS enforcements and shared-resource interference among collocated threads into account when making resource allocation decisions. Our solution uses a prediction module to estimate the latency of each computational module by employing a queue-based model and estimating of the future rates for incoming data records. We benchmarked the proposed solution against the enhanced round robin policy with respect to two major performance metrics of response time of data records processing (18.8% improvement), and QoS violation rates (77% overall improvement) for the workloads that either resemble Poisson distribution for arrival or heavy-tailed patterns (Weibull distribution).

The remainder of this paper is organized as follows. Section 3 concisely introduces the background knowledge to help the reader appreciate the corresponding concepts of the new paradigm. Section 4 gives insights into the proposed controller. Section 5 summarizes the results on the experimental evaluations, followed by some comparison to related work presented in Sect. 2. Finally, Sect. 6 draws some final conclusions.

2 Related Work

Parallel data-flow platforms have been effectively employed in the field of big data mining where algorithms show an iterative nature. Naiad [22] has been designed as the first distributed system for running parallel and iterative operations over either batch or streaming data-flows. In Naiad, each message has a logical time-stamp as well as some location-generation meta-data that allows the underlying system to figure out the right order and the associated priority of each message. However, the thread-level elasticity is not supported by the system.

Apache Spark [33] is a fast, in-memory data processing engine to execute iterative algorithms over the streaming data-sets. It probably has the most similarity with timely data-flow paradigm when compared to other existing frameworks. However, the big difference between the two paradigms is that while the timely data-flow engine maintains a set of persistent tasks that repeatedly send and receive the data records, the Apache Spark engine starts and stops worker tasks (by breaking the computation into lots of independent tasks) similar to most batch processing engines do. This allows the TDF engine to support very fast scheduling of operators in microsecond scales, as shown by [19].

Many existing resource allocation strategies, *e.g.*, [15,17] manage resources based on OS level metrics, such as per core utilization, I/O capacities, and energy usage of resources while ignore the negative performance caused by interference at the shared resources (LLC or memory bandwidth). However, a careful study

by [30] 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.

The work in [32] attempted to anticipate the micro-architecture-level interference by using an *offline* profiling phase. Rao *et al.* proposed an effective metric to predict the performance of applications running in a NUMA system [24]. Such a metric can be leveraged to design a resource allocation that is aware of contention among shared resources. However, obtaining such an interference signature through profiling might not be feasible in every practical cases, as the interference attributes of applications could change over the run-time. Authors in [7,8] proposed a method to reduce the negative impact of architecture-level shared resource contention on a hyper-visor-based cloud platform. However, it seems that these projects concern mainly with tuning resources on one server node, while our focus is to devise a resource allocation mechanism in a cluster of server nodes where *parallelization* of each data-flow application is of great importance to the overall performance of the system.

Using a predictive-based model is not new in computing systems [20,21]. [2,20,21,23]. Most of these works proposed a multi-input, multi-output (MIMO) resource controller that automatically adapts to dynamic changes in a shared infrastructure. Such models try to estimate the complex relationship between the application performance and the resource allocation, and adjusts the embedded model by measuring the clients' response time. While there are similarities between the proposed solution with previous MPC-based controllers, our solution responds to the degraded performance level by measuring the number of waiting messages and then applying a more accurate queuing based formula to estimate the response time of each application.

3 Background

Designing a scalable and fault-tolerant distributed system for running parallel programs for processing streaming data (or data-flow) has been recently receiving a lot of attention. This includes dealing with the upcoming issues in the processing of data-flow in near real-time fashion. This section provides brief background information about the core concepts used in this area, called *timely dataflow*, which is first introduced by Microsoft researchers in 2013 [22].

3.1 Cyclic Data-Flow Model

Timely data-flow proposes a new concept to embrace the three advantages of prevalent giant models for processing large amount of data. It offers (1) *high throughput* (for batch processing systems), (2) *low latency* (for stream processing engines), and (3) the ability to perform *iterative*, *stateful*, and *incremental* computations over incoming data-flow. This new model resolves the complexities of combining such features under one umbrella, *e.g.*, nesting loops inside streaming contexts that keeps state of the computations [22].

The model supports a new form of logical timestamps attached to each data-flow computation as an efficient, lightweight coordination mechanism for supporting iterative and incremental processing. At any given time, the platform maintains a set of point-stamps of those message which are still in progress. This is done to track the progress of messages processing. So, each parallel worker knows the number of outstanding data-flow messages that are still live and needs to be delivered for further processing. The directed data-flow graph allows the platform to efficiently track the set of data records that might possibly flow throughout the computational graph. Such information can be used to quickly coordinate among all working threads to detect the possibility of additional data records to arrive at future epochs or iterations [6].

3.2 Shared Resource Interference

To enhance the utilization of the computational resources, a service provider can employ the "consolidation" method to host multiple data-flow applications submitted by different end-users into one (or more) physical node(s). Nevertheless, one cannot ignore the performance degradation experienced by consolidated applications due to the fact that one application may evict the data of other collocated applications whenever it is context switched to a CPU core. The consequence is an undesirable increase in the latency of other applications to access their own data in the main memory in the next CPU cycles.

It is well known that finding an effective consolidation solution in a shared-memory platform is challenging [27,29,30], mainly because each application has its unique resource consumption attributes while at the same time it requires a certain amount of quality of service level to be guaranteed by the underlying platform. Furthermore, applications can compete with each other to access the shared micro-architecture level resources such as last level cache and/or memory bandwidth, while the incoming traffic of data-flow to each application can vary temporally. To the best of our knowledge, no empirical research study exists to address the problem of the negative performance impact of shared resource contention among collocated timely data-flow applications.

4 QoS-Aware CPU Cap Adjustment

To achieve the required performance level enforced by QoS rules, *e.g.*, end-to-end response time, we use a model based on queuing theory to adjust CPU caps of each sub-component. We also employ "control theory" principles to design a robust strategy that dynamically regulates the performance parameters of each sub-component in response to the continuous feedback from the state of the underlying platform. The key idea is to leave the resource allocation decision to run-time, in which the resource controller can measure the following performance metrics: (1) the incoming rate of each application, (2) the available capacity of CPU capacity per host, (3) the QoS violation rate per each application, and (4) the contention on shared resources per host.

To resist against the temporal changes in the arrival rate of each module, we adjust the percentage of the CPU core to allocate to each working thread according to the number of outstanding messages in the main buffer of each computational module which is a good approximation of the end-to-end delays that each data-flow message may experience. We use a model predictive control (MPC) as a mathematically well-defined mechanism (1) to predict the future average arrival rate of messages to each data-flow, and (2) to dynamically make CPU cap decision based on the current and the predicated future states. At each controlling epoch the controller measures a set of performance metrics and compares them with their desired value reflected in SLA contract. MPC-based controller can provide a robust performance despite the modelling errors [25] is to keep the tracking error within an acceptable range.

The proposed mechanism consists of four components of system model, estimator, optimizer, and anti-saturation. The system model uses a formula for $G/G/M$ queues to abstract the complex running time behaviour of each thread. A $G/G/M$ queue represents a system with M servers where both the interarrival times of customers and the relevant service times have a general distribution.

The estimator uses a simple formula based on auto regressive integrated moving average (ARIMA) model to predict the input traffic rate of each application. The optimizer module uses a dynamic programming method to iteratively adjust CPU cap of each thread by considering the performance values obtained from the other two modules. Lastly, the anti-saturation component is used to prevent over-utilization of CPU in each host, particularly in cases when the CPU demand is higher than the available capacity.

4.1 QoS Guarantee Semantic

We assume that there are exactly Q different classes of QoS contracts that an end-user can choose from. Each QoS class $1 \leq q \leq Q$ is indicated using a predefined pair of values, denoted by $\langle \omega_q^*, \mathcal{V}_q \rangle$, each reflects a fixed service parameter. The ω_q^* value defines the maximum acceptable average processing delay of messages belonging to an application of class q.

$\mathcal{V}_q(\Delta T)$ reflects an acceptable upper bound for the percentage of QoS violation incidents for all applications in class q during an arbitrary interval of size ΔT. A good candidate for $\mathcal{V}$ is a linear rule like $\mathcal{V}_q = 1 - \frac{q}{Q+C}$, where C is a constant. As a concrete example, assume a scenario that $|Q| = 3$, and the associated upper bounds of each QoS class is taken from are $\mathcal{V}_{q=1..3} \in \{0.99, 0.90, 0.70\}$, where the first class ($q = 1$) has the highest priority. So, the delay of processing data-flows belonging to q_1 can be higher than ω_1^* only for 1% of the entire messages entered to the system during any arbitrary interval.

4.2 System Model

We use Allen-Cunneen approximation of $G/G/M$ queue [4] to estimate an upper-bound of the *average* end-to-end response time experienced by each message in

the main buffer of each computational vertex. Based on this formula, the average waiting time of customers (denoted as W_M) in any general $G/G/M$ queue can be approximated by the following equation:

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

where $C_D = \sigma_D / E_D$ and $C_S = \sigma_S / E_S$ are the coefficients of variation for inter-arrival time and service time, respectively. Sometimes, the term $\frac{C_S^2 + C_D^2}{2}$ is referred to the *stochastic variability* of the queue. The term $P_{cb,M}$ is the probability that all servers are busy; hence, the waiting time of a recently arrived customer is above zero. For a queuing system with only one server ($M = 1$) this parameter can be calculated as $P_{cb,M} = \rho_{C_i}$, where $\rho_{C_i} = \frac{\lambda_{C_i}}{\bar{\mu}_{C_i}}$ is the service traffic intensity of component C_i (*i.e.*, its utilization). Here, λ_{C_i} is the average number of messages arriving to the main buffer of component C_i per unit of time, and $\bar{\mu}_{C_i}$ is the average number of messages to be served per unit of time by each working thread associated with component C_i. Otherwise, *i.e.*, $M \geq 2$, one can use the following formula as suggested by [5].

$$P_{cb,M} \approx \begin{cases} (\rho^M + \rho)/2 & \text{if } \rho \geq 0.7 \\ \rho^{\frac{M+1}{2}} & \text{otherwise} \end{cases}, \tag{2}$$

where $\rho_{C_i} = \frac{\lambda_{C_i}}{M \bar{\mu}_{C_i}}$. It is worth noting that while the A-C formula was developed using some computational-based estimation techniques without a formal proof, it gives a very good approximation to the average waiting time of customers in a $G/G/M$ queuing system. As reported by Tanner in [28], the value obtained by the A-C formula were within 10% of their actual values in most scenarios.

At any given time, each computational module needs to identify the right number of concurrent working threads, *i.e.*, the parallel degree, shown by M in (1) and (2). To make the problem tractable, we allow each computational module to increase or decrease its parallel degree at most by *one* during every controlling interval. Let $M^o_{(\tau,C_i)}$ denote the parallel degree of a computational module C_i during a given interval τ. The controller just needs to recompute (1) and (2) for the subsequent intervals by replacing the parallel degree with $M^o_{(\tau,C_i)}$, $M^+_{(\tau,C_i)} = M^o_{(\tau,C_i)} + 1$ and $M^-_{(\tau,C_i)} = M^o_{(\tau,C_i)} - 1$, respectively, and then choose the best result among them.

If a computational module resides within a loop context, the average waiting time obtained by (1) needs to be multiplied by the average number of times that the loop context is run over its input messages. Let $\bar{v}_{C_i}$ denote the average value of the loop variable for C_i as a computational module within a loop context. Finding the exact value of $\bar{v}_{C_i}$ is computationally expensive, as one needs to keep track of all messages processed by each computational module. A good estimation of the loop variable for each module perfectly works in most practical scenarios, as the MPC-based scheme is not too sensitive to the correctness of its input values.

To this end, we employ an estimation procedure based on a well-known Monte Carlo sampling method, called the *AA* Algorithm [10]. This algorithm is a fully-polynomial randomized approximation scheme (FPRAS) that uses the *minimum* possible number of measurement to estimate the value of $\bar{v}$.

Using the *AA* Algorithm, we can drive a good estimation of the total average processing time of messages that belong to an application by summing up the response time of a vertex (multiplied by $\bar{v}_{C_i}$ if C_i resides in a loop context).

We use ARIMA model to estimate the average arrival rate of messages to the first computational component of each data-flow application for the next controlling intervals. Based on this model, the future value of a random variable, such as $\lambda_{\kappa}^{C_1}$, can be estimated using a series of previous observations [9]. The controller then can calculate the desirable amount of CPU capacity to be assigned to each thread such that the application meets its QoS requirements.

4.3 Resource Allocation Parameters

The optimization module operates in periodic control intervals to adjust the CPU cap for each data-flow application. The optimization goal is accomplished through a two-phase process. First, it determines the desirable demand of each application for the CPU credit. Then, it computes the resource share that can be allocated to each application based on available resource capacity, a cost-benefit analysis, and the server nodes' status. Upon receiving of initial desirable CPU demands from all applications, the optimization module determines the possibility of satisfying all demands by considering the fact that there might not be enough resource capacities available within the entire cluster. This enables all applications to meet their performance targets as specified by the QoS requirements.

In case of *resource scarcity*, however, the optimization module tries to maximize the contribution (or the reward) that the system provider receives from a resource allocation decision by applying a cost-benefit analysis. Each working process has been assigned a quantum-based *cap*. A non-zero cap means that the amount of CPU time to be assigned to the thread process cannot run above the certain cap amount (even if the other processes are idle). So, a cap value of 200 means two CPU cores and 50 means half a core [1].

4.4 Optimization Module

We define a contribution function to reflect that quantifies the value of the gains (and losses) for all applications affected by a CPU adjustment decision. Let $D^*_{a_i,\tau}$ denote the CPU cap demanded by a particular application a_i at any given time τ. Let R^o_τ denote the total amount of CPU cap that the available switched-on server nodes can provide. Further, let R^+_τ, and R^-_τ denote the provided CPU cap if one server is added or removed from the current set of switched-on server nodes, respectively.

To mitigate the negative effect of poor RA decisions on the overall revenue, we only calculate the R^+ and R^- at any controlling epoch. We will allow the

controller to add or remove at most one server node from the available server nodes at any decision interval; hence, the feedback loop can detect the negative effect of any poor decision and let the controller stop such results.

Let us define a *contribution* function for each application a_i, denoted by $\mathcal{C}_{a_i}(r_{a_i})$, that determines the reward that is received by the service provider if r_{a_i} CPU cap is allocated to this application as follows.

$$\mathcal{C}_{a_i}(r_{a_i}) = \mathcal{I}(q_{a_i}) \times (r_{a_i} - D^*_{a_i}), \tag{3}$$

where q_{a_i} is the QoS class that the application belongs to, and $\mathcal{I}(q_{a_i})$ represents the importance weight associated with each QoS class. At any given controlling interval τ, we would like to maximize the total contribution of the service provider as $\max_r \sum_{a_i \in \mathcal{A}} \mathcal{C}_{a_i}(r_{a_i})$, where $\mathcal{A}$ denotes the set of available applications. We solve the aforementioned optimization problem with subject to the obvious constraint of $r_{a_i} \geq 0$ and another constraint on the available resource cap on three different cases (*i.e.*, R^o, R^+ and R^-).

$$\sum_{a_i \in \mathcal{A}} r_{a_i} = R^*_\tau \text{ where } R^*_\tau \in \{R^o_\tau, R^+_\tau, R^-_\tau\}. \tag{4}$$

We then pick the best solution among the three cases. If we assume that r_{a_i} can be only taken from discrete values, *e.g.*, $r_{a_i} \in \{10, 20, \cdots\}$, then solving this problem can be done using a standard dynamic programming strategy as follows. Let $V_i(R_i)$ denote the value of having R_i resource cap remaining to allocate to any application a_j where $j \geq i$. So, we only need to recursively solve the following Bellman's equation:

$$V_i(R_i) = \max_{0 \leq r_i \leq R_i} \left(\mathcal{C}_{a_i}(r_{a_i}) + V_{i+1}(R_i - r_{a_i})\right). \tag{5}$$

Let $n = |\mathcal{A}|$ denote the total number of applications. The initial step is to solve the problem of $V_n(R) = \max_{0 \leq r_{a_n} \leq R^*} \mathcal{C}_{a_n}(r_{a_n})$, for all possible values of $0 \leq R \leq R^*_\tau$.

To quantify the slowdown rate caused by a consolidation action, we pursue an effective method based on the solution initially introduced in [27,30]. So, the impact of workloads' contention on both LLC and memory bandwidth can be computed as a sudden rise in the *m*emory *b*and*w*idth *util*ization, denoted by MBW_{util}. By measuring the two standard hardware events as an indicator of memory reads and writes, one can compute the utilization level of memory bandwidth [27,30] (using *perf* in Linux).

5 Experimental Evaluation

We built a proof-of-concept prototype using a modular open-source implementation of timely data-flow in Rust (the source code is obtainable from [18]). We performed a set of experiments using synthetic applications to validate the versatility of the proposed solution under sudden changes in the arrival rate of

data-flow applications. We measure the effectiveness of the proposed solution with respect to the following metrics: (1) the average latency experienced by each data-flow application, and (2) the amount of QoS violations experienced by each data-flow application.

We compare our solution against an enhanced round robin method (ERR) which assigns a fixed value for the number of working threads determined by the QoS class that the application belongs to. We fixed the number of QoS classes to three and the parallel degree values to $\{8, 4, 2\}$ to be used by ERR heuristic. The enhanced interference-aware version of this strategy averts sending extra load to a physical machine that is marked as over-utilized.

All of the experiments reported in the following sections have been performed in a local cluster consisting of 4 nodes (from Amazon EC2) with total 16 logical cores. Each machine is installed with 8 GB of main memory and equipped with a 3.40 GHz Intel i3 CPU. The controller developed in C++ uses a dedicated node equipped with 2.3 GHz CPU with 16 GB of RAM.

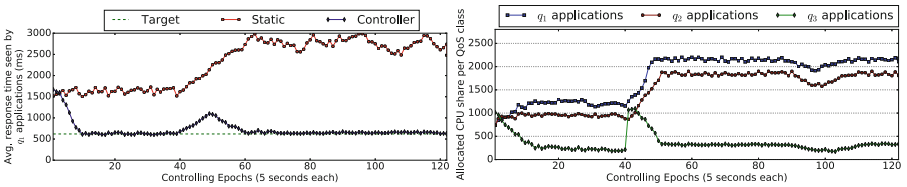
5.1 Attributes of Synthetic Applications

We created $|\mathcal{A}| = \{50, 100, 200\}$ different data-flow applications where each application has four computational modules. Each computational module runs a CPU-intensive script (taken from RUBiS benchmark, a well-known cloud web application that emulates the core functionality of an auction site) that its running time varies based on the type of the incoming message ranging from 100 ms to 3400 ms with an average of 900 ms. We select three different QoS classes in our experiments and randomly assign each application to one of the QoS classes, where the associated upper bound of each class is $\mathscr{V}_{q=1..3} \in \{0.99, 0.90, 0.50\}$.

We bind the first computational module of each application to an external message emitter where its generation rate is a *varied* value taken from either a Poisson or a Weibull distribution. The corresponding parameter in Poisson case varies in range of $\lambda_P \in [0.2, 1]$, where λ_P represents the average number of messages generated per hundred milliseconds. The Weibull distribution occurs often in applications with heavy-tailed patterns. We allow the two corresponding parameters in Weibull case, *i.e.*, α_W as the scale and β_W as the shape parameter, to vary as $\alpha_W \in [1.1, 4]$ and $\beta_W \in [2, 6]$. The average number of incoming data elements per hundred milliseconds in the Weibull case can be derived by $\alpha_W \Gamma(1+1/\beta_W)$ (Γ: Gamma function). The controller also uses a history window of 3 past intervals prior to the current epoch.

Figure 1(a) represents the 99^{th} percentile average response time of applications belong to the highest priority class (q_1) as a function of time (controlling epochs) when the QoS target sets to 620 ms to achieve. Initially all applications have the same CPU cap allocations. Compared to the static allocation strategy, the proposed scheme can co-ordinate CPU cap adjustment based on the QoS requirements in the run-time. It automatically adjusts the CPU cap of q_1 applications to reduce their response time from 1500 ms close to the target value (by augments their initial CPU cap by a factor of 3.4×).

In fact, we dynamically modify the workload stress of some applications in a way that there are not enough CPU shares to comply with the performance targets of all applications as follows. During the first 40 control epochs (*Phase I*), there are enough CPU caps so that all applications belonging to different QoS classes can meet the requested performance target. But at this time (and continuously toward the last epoch) (*Phase II*), we intentionally allow applications from different QoS classes increase their message generation rates by a factor of 1.8×, 2.7×, and 3.6× for q_1, q_3, and q_3 classes, respectively, in a linear fashion from 40^{th} epoch till 51^{th} epoch. The generation rates remain still toward the end of the experiment for all QoS classes.



(a) (b)

Fig. 1. (a) Improvement in average response time compared to the static allocation scheme, and (b) CPU allocation share for applications in different quality of service classes. An unexpected burst of messages arrives to the system at 40-th epoch (a majority of the burst messages (>75%) belongs to the lowest QoS class, q_3).

In the second phase, the controller cannot fully satisfy all incoming demands; hence, it runs the cost/benefit of compromising among the performance level of different applications. So, it decides to assign more CPU shares to q_1 applications; hence, force their response time converge to the target faster than the q_2 and q_3 applications. As a result, it refrains from satisfying almost all (87%) and some (33%) of the demands from q_3 and q_2 applications, respectively, to meet the requested target value during the burst period, when the target is set to be 1400 ms and 1100 ms, respectively.

The static scheme tends to equally (or based on the incoming workload of applications) distribute CPU shares among applications which in most scenarios can cause a performance degradation (*i.e.*, QoS violations) for both q_1 and q_2 applications ($> 91\%$ and $> 74\%$ during the run-time when the target is set to be 900 ms and 1200 ms, respectively). Nevertheless, the issue can be amplified (by a factor of up to 3×) due to the resource scarcity during the burst periods.

Figure 1(b) depicts the amount of CPU caps assigned by our solution to different applications belonging different QoS classes. It confirms that the controller exhibits a fast convergence for adjusting CPU caps for applications of highest QoS classes to satisfy their demands. During Phase II the controller decides to add a new server node to the existing cluster as it recognizes that the current computing capacity is not enough to satisfy all demands from q_1 applications.

On the other hand, the static ERR scheme over-provisions CPU cap for q_3 applications severely diminishes achieving the performance target for both q_1 and q_2 applications ($> 91\%$ and $> 77\%$ on average, respectively).

Table 1. Improvement in (a) average latency [%] of processing of data records, and (b) average reduction in QoS violation incidents [%] (during Phase II) achieved by the proposed solution compared to ERR in different scenarios.

Application's QoS class	Poisson (θ)		Weibull (α, β)		Avg.
	1	0.2	(1.1, 4)	(4, 6)	
q_1	14.2	18.1	13.6	29.3	18.8
q_2	6.8	12.3	9.0	21.7	12.45
q_3	1.1	-1.9	1.2	-6.5	-1.5
Average	7.3	9.5	8.0	14.8	9.9

(a)

Application's QoS class	Poisson (θ)		Weibull (α, β)		Avg.
	1	0.2	(1.1, 4)	(4, 6)	
q_1	89	96	178	227	147.5
q_2	75	80	103	98	89
q_3	14	3	-7	-28	-4.5
Average	59	60	91	99	77

(b)

Table 1(a) lists the improvement in average processing time per data records experienced by each application grouped by the corresponding QoS class. The total number of data-flow applications in this scenario is fixed to 200. Modifying the arrival distribution of data records affects the performance of the proposed approach in reducing the overall average processing time of applications with highest QoS requirements (*i.e.*, q_1 and q_2). Particularly, such an improvement is more significant in the Weibull distribution of incoming traffic (which can be considered as a heavy-tailed workload) by an average of 21.5% (max 29.3%).

Table 1(b) lists the amount of reduction in the QoS violation incidents experienced by applications in different QoS classes that is achieved by applying our solution compared to the outcome of the enhanced round robin scheme in different scenarios. Our solution can reduce the QoS violation incidents on average by near 78% (maximum 227%) compared to the ERR heuristic that uses all available computing resources. Particularly, the improvement in reducing QoS violation of applications in highest QoS classes is more significant when the incoming traffic follows a heavy-tailed (Weibull) pattern. The average of such improvement in such cases is 203%.

6 Conclusions

Designing a well-utilized CPU cap adjustment strategy for timely data-flow platform requires understanding the dynamic functioning of computational modules in a shared platform. Timely data-flow is a powerful and general-purpose programming abstraction for creating iterative and streaming computational components that no other existing system (such as streaming/batch processing engines) supports. While the timely data-flow programming model supports thread-level parallelism as form of thread communication via ordered messages at scale, it is an absolute requirement to design an elastic CPU cap adjustment algorithm that continually monitors the related performance metrics of

the underlying system to assign the right amount of CPU capacity to applications that might request different QoS level. In a shared distributed environment, such non-cooperative applications fiercely compete for obtaining shared resources (such as last level cache) at the cost of performance degradation of each other.

An uncontrolled resource allocation discipline along with the uncoordinated execution of each computational component can severely damage the overall QoS fulfillment, by not hitting the performance target. In this paper, we presented a low-overhead feedback-driven resource allocation mechanism that dynamically adapts computational resources for co-running timely data-flow applications in a shared cluster. It consists of a model predictive based controller that adjust the resource share of each application by solving an optimization problem using a dynamic programming method to fulfil application's SLO. The effectiveness of the proposed solution has demonstrated an average improvement of performance in terms of latency of processing data records for applications in high QoS classes by 21% in average compared to the enhanced round robin policy.

Future Work. We realized that the proposed controller has a certain upper bound on achieving its performance when running on a local cluster, particularly when a majority of computing modules suddenly receives a huge traffic. In such cases, the proposed controller needs to be equipped with a migration technique to launch more threads to stop further QoS violation. The next step can be a comprehensive study for comparing the effectiveness of the proposed method with some advanced sophisticated scheduling algorithms.

Acknowledgement. We would like to acknowledge the support by Australian Research Council (ARC) for the work carried out in this paper, under Linkage project scheme (LP160100406). Samee U. Khan's work is supported by (while serving at) the National Science Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Xen credit scheduler. wiki.xen.org/wiki/Credit_Scheduler. Accessed 1 Nov 2017
2. Abdelwahed, S., et al.: On the application of MPC techniques for adaptive performance management of computing systems. IEEE Trans. Netw. Serv. Manag. **6**(4), 212–225 (2009)
3. Akidau, T., Balikov, A., et al.: Millwheel: fault-tolerant stream processing at internet scale. Proc. VLDB Endow. **6**(11), 1033–1044 (2013)
4. Allen, A.O.: Probability, Statistics, and Queueing Theory. Academic Press, Cambridge (2014)
5. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S.: Queueing Networks and Markov Chains. Wiley, Hoboken (2006)
6. Thekkath, C.: Naiad project (2017). https://www.microsoft.com/en-us/research/project/naiad

7. Chen, L., Shen, H.: Considering resource demand misalignments to reduce resource over-provisioning in cloud. In: IEEE Conference on Computer Communications (2017)
8. Chen, L., Shen, H., Platt, S.: Cache contention aware VM placement & migration in cloud. In: International Conference on Network Protocols, pp. 1–10. IEEE (2016)
9. Croarkin, C., Tobias, P., Filliben, J.J., Hembree, B., Guthrie, W.: NIST/SEMATECH e-Handbook of Statistical Methods. NIST, U.S. Department of Commerce, NY, USA (2006). http://www.itl.nist.gov/div898/handbook
10. Dagum, P., Karp, R., Luby, M., Ross, S.: An optimal algorithm for Monte Carlo estimation. SIAM J. Comput. **29**(5), 1484–1496 (2000)
11. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
12. Murray, D.: An introduction to timely dataflow (2017). bigdataatsvc.wordpress.com/2013/09/18/an-introduction-to-timely-dataflow/
13. Dudoladov, S., Xu, C., et al.: Optimistic recovery for iterative dataflows in action. In: ACM SIGMOD International Conference on Management of Data, pp. 1439–1443 (2015)
14. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. ACM Comput. Surv. (CSUR) **46**(4), 46 (2014)
15. Huang, X., Xue, G., Yu, R., Leng, S.: Joint scheduling and beamforming coordination in cloud radio access networks with qos guarantees. IEEE Trans. Veh. Technol. **65**(7), 5449–5460 (2016)
16. Li, B., Diao, Y., Shenoy, P.: Supporting scalable analytics with latency constraints. Proc. VLDB Endow. **8**(11), 1166–1177 (2015)
17. Li, K., Liu, C., Li, K.: An approximation algorithm based on game theory for scheduling simple linear deteriorating jobs. Theor. Comput. Sci. **543**, 46–51 (2014)
18. McSherry, F.: A modular implementation of timely dataflow in rust. https://github.com/frankmcsherry/timely-dataflow. Accessed 1 Nov 2017
19. McSherry, F., Isard, M., et al.: Scalability! but at what cost? In: HotOS (2015)
20. Mencagli, G.: Adaptive model predictive control of autonomic distributed parallel computations with variable horizons and switching costs. Concurrency Comput.: Pract. Exp. **28**(7), 2187–2212 (2016)
21. Mencagli, G., Vanneschi, M., Vespa, E.: A cooperative predictive control approach to improve the reconfiguration stability of adaptive distributed parallel applications. ACM Trans. Auton. Adapt. Syst. **9**(1), 2 (2014)
22. Murray, D.G., McSherry, F., et al.: Naiad: a timely dataflow system. In: ACM Symposium on Operating Systems Principles, pp. 439–455 (2013)
23. Padala, P., et al.: Automated control of multiple virtualized resources. In: European Conference on Computer Systems (EuroSys), pp. 13–26. ACM (2009)
24. Rao, J., Zhou, X.: Towards fair and efficient SMP VM scheduling. In: SIGPLAN Symposium on Principles & Practice of Parallel Programming, pp. 273–286. ACM (2014)
25. Rawlings, J.B., Mayne, D.Q.: Model Predictive Control: Theory and Design. Nob Hill Publishing, LLC, Madison (2009)
26. Şahin, S.: C-stream: a coroutune-based elastic stream processing engine. Ph.D. thesis, Bilkent University (2015)
27. Subramanian, L., Seshadri, V., Ghosh, A., Khan, S., Mutlu, O.: The application slowdown model. In: Microarchitecture (MICRO), pp. 62–75. IEEE (2015)
28. Tanner, M.: Practical Queueing Analysis. McGraw-Hill, New York City (1995)
29. Tembey, P., Gavrilovska, A., et al.: Application & platform-aware RA in consolidated systems. In: Symposium on Cloud Computing, pp. 1–14. ACM (2014)

30. Wang, H., Isci, C., Subramanian, L., Choi, J., Qian, D., Mutlu, O.: A-DRM: architecture-aware distributed resource management of virtualized clusters. ACM SIGPLAN Not. **50**(7), 93–106 (2015)
31. Yang, F., Qian, Z., Chen, X., Beschastnikh, I., Zhuang, L., Zhou, L., Shen, J.: Sonora: a platform for continuous mobile-cloud computing. Technical report, Microsoft Research Asia (2012)
32. Ye, K., et al.: Profiling-based workload consolidation & migration in VDCs. IEEE Trans. Parallel Distrib. Syst. **26**(3), 878–890 (2015)
33. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. HotCloud **10**(10–10), 95 (2010)