Efficient Stream Processing in the Cloud

Dung Vu[1], Vana Kalogeraki[2], and Yannis Drougas[3]

[1] Department of Computer Science and Engineering,
University of California-Riverside, USA
dungv@cs.ucr.edu
[2] Department of Informatics, Athens University of Economics and Business, Greece
vana@aueb.gr
[3] Environmental Systems Research Institute, Redlands, USA
drougas@esri.com

Abstract. In the recent years, many emerging on-line data analysis applications require real-time delivery of the streaming data while dealing with unpredictable increase in the volume of data. In this paper we propose a novel approach for efficient stream processing of bursts in the Cloud. Our approach uses two queues to schedule requests pending execution. When bursts occur, incoming requests that exceed maximum processing capacity of the node, instead of being dropped, are diverted to a secondary queue. Requests in the secondary queue are concurrently scheduled with the primary queue, so that they can be immediately executed whenever the node has any processing power unused as the results of burst fluctuations. With this mechanism, processing power of nodes is fully utilized and the bursts are efficiently accommodated. Our experimental results illustrate the efficiency of our approach.

Keywords: Stream Processing, Peer-to-Peer, Distributed Systems.

1 Introduction

Over the years, we have experienced the proliferation of distributed stream processing systems that deal with large volume and high-rate data streams. A number of stream processing systems have been developed, including Aurora [2], STREAM [3], TelegraphCQ [1] and Cougar [4]. These systems are characterized by continuous, large-volume, high-rate data streams that are generated by geographically distributed sources and processed concurrently and asynchronously by one or more processing components (*e.g.,* filtering operations aggregation operators, or top-K querying) to perform various tasks such as IP network traffic monitoring and analysis for detecting DoS attacks, location tracking, text mining, financial data analysis, multimedia delivery, and outlier detection in sensor networks [3,1]. More recently, the *cloud computing* model promotes the development of an infrastructure comprising large groups of servers that enables the sharing of computational, storage and network resources rather than having dedicated servers and personal commodity machines to run the applications.

X. Zhang and D. Qiao (Eds.): QShine 2010, LNICST 74, pp. 265–281, 2011.

Such an infrastructure has important flexibility, scalability and economical advantages. This model has been applied successfully by a number of companies such as Amazon, IBM and Google [15] [16].

In such an environment, burst management becomes an important challenge to provide real-time delivery of the streaming data while dealing with the sharing of the computing resources and the unpredictable increase in the volume of data. Data streams that arrive in bursts may create overloads at the processing and networking resources, causing losses of critical data and severely affecting the application performance. The problem is challenging because of the sharing of the same computing and network resources by multiple competing applications, combined with the short duration and unpredictability of the occurrence of the bursts. In [5] the authors report, that, to guarantee the application response time for an increase from 90% to 100% of bursts in storage systems, the corresponding resource capacity needs to increase from 4 to 7 times. Even a small burst increase from 99.9% to 100%, which is only accounted for the last 0.1% of the bursts, the capacity needs to increase by a factor of 2.4 times. As a result, attempts to accommodate 100% of bursts is very expensive and sometimes inpractical.

We believe that careful scheduling of the execution of the data streams on the system resources can maximize the probability that the timing requirements of the applications are met and furthermore reduce resource costs. Scheduling policies that decide the processing ordering of the data streams on the system resources where bursts occur, can greatly assist in compensating for delays. However, scheduling distributed streaming applications brings additional challenges. The scheduling algorithm must be distributed since the streaming applications invoke components concurrently and asynchronously on multiple nodes and the occurrence of a burst is not confined into a single node, rather it affects multiple nodes running components of the streaming applications. Furthermore, the scheduling algorithm should adapt dynamically to changes in the application behavior and maintain accurate resource measurements, since the burst produced by one application can cause queuing delays experienced in the execution of other distributed streaming applications in the system.

In this paper we address the problem of scheduling distributed data streams to efficiently accommodate bursts in distributed stream processing systems. Our main idea is to dynamically control the execution of the applications and delay the processing of bursty data streams when resource capacities are exceeded. Our approach is as follows: we implement a primary and a secondary queue at the Scheduler component of each node, to store data streams pending execution. Requests in the primary queue are scheduled to meet their timing requirements while the ones in the secondary queue are served with best effort. When the rate of the incoming requests exceeds a node's processing capacity and the Scheduler's primary queue is full, requests of the excess rate, instead of being dropped, are diverted to the secondary queue. When there is space in the primary queue due to temporarily reduced rates, requests from the secondary queue are brought back to the primary queue and scheduled for execution. The advantage of our approach, is that, by selectively delaying the execution of the bursty applications,

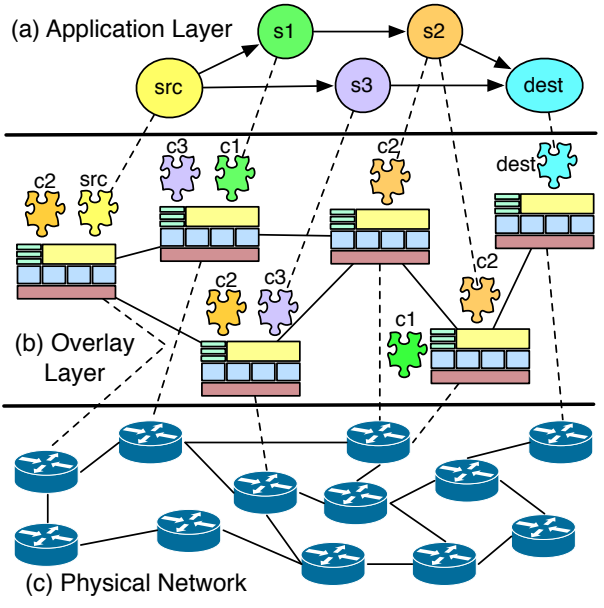

Fig. 1. The architecture of our distributed stream processing system

we can effectively reduce the effects of the bursts. With this mechanism, the effect of the bursts is confined only to the bursty applications while missed application deadlines are reduced to a minimum. Furthermore, by considering queuing delays when scheduling the requests on the system resources, applications that are projected to miss their deadlines are dropped early on. Detailed experimental results of our proposed mechanism over our distributed stream processing system Synergy [6] demonstrate the efficiency and benefits of our approach.

2 System Model and Problem Formulation

2.1 Scheduling Model

Each application app_q is represented as an acyclic graph that consists of a sequence of the services to be invoked. The services are distributed across multiple peers in the system; the distributed stream processing application is executed collaboratively by the peers of the system that host the corresponding services. The execution of an application in the system is triggered by a user request; this will trigger the instantiation of all the services that comprise the application. The instantiation of a service on a node is called *component*. A component operates on collections of tuples generated by a data source, called *data units* (or application data units - ADUs). Examples of data units are sequences of picture or audio frames (for example, in a multimedia application), or sets of measured values (for example, in a sensor data analysis application). The size of a data

unit depends on the application. Upon the issue of a user request, our system discovers and instantiates the appropriate components on the system nodes to perform the processing required by the application.

The streaming applications are aperiodic and their arrival times is not known a priori. Each application app_q is characterized by the following parameters: (i) Deadline D_q is the time interval, starting from the time the user submits the request, within which the application must complete execution. (ii) Rate r_q represents the delivery rate of the data units of the application as requested by the user. (iii) Projected Execution Time $Proj\_Exec_q$ is the estimated amount of time required for the distributed streaming application to complete. The projected execution time of the application includes the processing times of the components comprising the application, the queuing times at the Schedulers' queues and the corresponding communication times. The difference between the $Deadline_q$ and the $Proj\_Exec_q$ is called the $Laxity_q$ of the application and represents a measure of urgency of the application. Upon reception of a data unit at a node, the data unit is inserted into the Scheduler's queue waiting to be processed. The order imposed on the data units on the Scheduler's queue depend on the scheduling algorithm implemented in the system.

Each component is characterized by the following parameters: (i) Processing time τ_{c_i} is the processing time required for component c_i to execute locally on the node and (ii) σ_{c_i} is the queuing time on the Scheduler's queue. We denote as (iii) r_{c_i} the incoming rate of the component c_i and (iv) sel^{c_i} be the selectivity of component c_i; this represents the ratio of output rate to input rate for the component. (v) Finally, we let $u_j^{c_i}$ represent the resource requirements of component c_i for resource j (where resource j can represent CPU, memory or bandwidth). Note that the resource requirements and component selectivity characteristics can be provided by the user prior to application execution or acquired through profiling at run-time with low-overhead.

We have implemented our approach in Synergy, our peer-to-peer stream processing system [6]. Synergy is middleware whose goal is to support the execution of distributed stream processing applications with QoS requirements. Each node in Synergy consists of the following modules: (i) a discovery module, that runs over the Pastry DHT, which is responsible for discovering components at runtime with low-overhead, (ii) a composition module is responsible for running a component composition algorithm to dynamically select components to configure and instantiate distributed stream processing applications, (iii) a monitoring module that is responsible for maintaining current resource availability, and, (iv) a routing module that routes data streams between Synergy nodes. We have extended Synergy with a scheduling component that implements our scheduling approach (described in this paper). Synergy's architecture is shown in Figure 1.

2.2 Problem Formulation

The objective of our Scheduling approach is as follows: Given a number of Q applications submitted into the system, our goal is to minimize the number

of applications that miss their deadlines. We present our problem formulation below:

The system must satisfy the following conditions: node and link capacity constraints, flow conservation constraint, and deadline constraint.

To satisfy the node and link capacity constraints, the sum of the utilization $u_j^{c_i}$ of resource j $(1 \leq j \leq J)$ made by all components c_i in a node n must not exceed the total resource availability U_j^n of resource j on that node as follows:

$$\forall n \in \mathcal{N}, \sum_{c_i \in n} r_{c_i} \cdot u_j^{c_i} \leq U_j^n, 1 \leq j \leq J \tag{1}$$

where r_{c_i} and $u_j^{c_i}$ are the assigned rate and unit resource needs for component c_i respectively.

To satisfy the flow conservation constraint, the output rate of a component c_i is computed based on the input rate of the component and the *selectivity* of the component sel^{c_i}. The selectivity of each component depends on the service of the component. Let $\mathcal{O}(c_i)$ be the set of downstream components of component c_i. Thus, the flow conservation constraint is given by:

$$\forall c_i, \sum_{c_j \in \mathcal{O}(c_i)} r_{c_j} = sel^{c_i} \cdot r_{c_i} \tag{2}$$

To satisfy the deadline constraint, the application q must finish execution before its deadline $\mathcal{D}_q$. Let $\mathcal{C}_j^q$ be a subset of components for application q. The execution time of the application $\mathcal{C}_j^q$ is the sum of the processing times τ_{c_i} of each component c_i invoked by the application, the queuing time σ_{c_i} experienced by each component c_i waiting at the Scheduler's queue for other components in the same node to execute, and the corresponding communication times δ_{c_i} between two adjacent components c_i and c_{i+1}. Since the end-to-end execution time of the application must be smaller than its deadline, the deadline constraint for the application q is:

$$\forall q \in \mathcal{Q}\ , \forall \mathcal{C}_j^q \in \mathcal{C}^q,$$

$$\sum_{c_i \in \mathcal{C}_j^q} \tau_{c_i} \ + \sum_{c_i \in \mathcal{C}_j^q} \delta_{c_i} \ + \sum_{c_i \in \mathcal{C}_j^q} \sigma_{c_i} \ \leq \ \mathcal{D}_q \tag{3}$$

Let r_{c_i} be processing rate of component c_i, then the processing time τ_{c_i} of component c_i is: $\tau_{c_i} = \frac{1}{r_{c_i}}$. Then, the deadline constraint becomes as follows:

$$\forall q \in \mathcal{Q}, \forall \mathcal{C}_j^q \in \mathcal{C}^q,$$

$$\sum_{c_i \in \mathcal{C}_j^q} \left(\frac{1}{r_{c_i}} + \delta_{c_i} + \sigma_{c_i} \right) \leq \mathcal{D}_q \tag{4}$$

Our goal is to find the excess load that exceeds the capacity of the primary scheduling queue and divert this load to a secondary queue, so that we meet the

node capacity and also minimize the effect on the currently scheduled data units as follows:

- Find excess load.
- Delay the execution of the excess load. Whatever excess load exists,
- We schedule it later using the least-laxity scheduling algorithm so that we minimize the deadline misses.

Let $r_{c_i}^{Max}$ be the maximum input rate that component c_i can admit without overloading, and $r_{c_i}^{Burst}$ be the burst rate that exceeds component c_i's maximum allowed input rate. Then, $r_{c_i}^{Excess} = r_{c_i}^{Burst} - r_{c_i}^{Max}$ is the excess rate of component c_i. The number of missed data units of a component is directly related to its excess rate. In addition, the excess rate of a component has direct impact on drops of other components in the node. This will be further discussed in section 2.3. Our scheduling algorithm aims to reduce data unit drops by minimizing the effects of excess rate of individual components.

2.3 Laxity-Based Scheduling

In this section, we discuss our Least-Laxity Scheduling (LLS) scheduling technique under bursty input rates. LLS has been successfully employed in distributed real-time systems such as in [7]. In least-laxity scheduling, each data unit is associated with a laxity value that represents a measure of urgency for the data units; these will be ordered in the scheduler's queue based on their laxity values. We compute the laxity value L_q of an application q as the difference between the deadline and the end-to-end projected execution time of the application:

$$L_q = Deadline_q - Proj\_Exec_q \tag{5}$$

The application with the smallest laxity value has the highest priority in the system. The laxity value for each data unit of the application is computed initially at the sources and is adjusted as it gets propagated in the distributed system, based on actual processing and network conditions. The purpose of this ordering of the data units is to allow for compensation for delays that were introduced at previous nodes. Especially in the presence of multiple distributed stream applications, the introduction of a new application may cause existing applications to experience higher delays due to queuing. If a data unit is delayed at the node's queue, its laxity value will diminish and thus its priority will increase. Streaming applications with negative laxity values are estimated to miss their deadlines, thus they are dropped from the queue. This approach allows us to implement a distributed scheduling algorithm that executes across multiple nodes and dynamically adapts to the current load conditions.

In least laxity scheduling, two data units belonging to different streaming applications arriving at the same node and having the same laxity value will be treated the same. When two data units have the same laxity value, their relative order in the queue is their arrival order. We finally note that out-of-order transmission of data units of the same application may potentially happen

in the case that a packet of a flow was delayed enough locally at a queue that its laxity value became smaller than the laxity value of an earlier data unit from the same application. Data units with smaller laxity values are still treated as more urgent. When data units of the same flow arrive at the destination, they can be post-ordered based on their id-numbers.

3 Our Two-Queue Scheduling Approach

In this section we present the operation of our approach. Our solution accommodates bursts by dividing the data units to be processed into two separate queues: A *primary* queue Q_1 and a *secondary* queue Q_2. Data units in Q_1 are scheduled based on their timing requirements. The ones in Q_2 are only guaranteed best-effort service: Data units in Q_2 will only be serviced when there are resources available on the node. This way we avoid over-penalizing the rest of the data units that are already scheduled for execution: Should there be enough idle time after the burst, they will also be processed.

The primary queue Q_1 is characterized by its predefined maximum size, $\max(Q_1)$. Data units in Q_1 are scheduled to be executed by their deadline. On the contrary, data units are put in Q_2 when the system is considered to be overloaded (Q_1 is full). Data units in Q_2 are executed in a best-effort manner. This means that they will be processed on the earliest slack of the system. $\max(Q_1)$ is defined based on the average processing time and the average laxity of the the data units on the node. More formally:

$$\max(Q_1) = \frac{1}{|C_n|} \sum_{i \in n} \frac{1}{r_{c_i}} \cdot \frac{1}{\tau_{c_i}}$$

where C_n is the set of components running on node n and τ_{c_i} is the average processing time of component c_i. The computation of $\max(Q_1)$ is based on the same logic as in [5]. The average input rate r_{c_i} of each component c_i is constantly monitored during system operation. The average processing time τ_{c_i} for a component c_i is extracted through profiling.

Under stable conditions, when there is no burst, the combined rates of all streaming applications do not exceed the system capacity. In this case, the ADUs are processed and propagated only through primary queues from source to the destination. No ADUs enter the secondary queue.

When a burst occurs, this means that the combined incoming rates of all streaming applications exceeds system capacity. As a result, the primary queues of overloaded nodes have become full. The excess load is then diverted to the secondary queues and kept there in least laxity order. If at any time during execution the laxity values become negative, the corresponding ADUs are dropped.

This way, incoming requests with excessive rates will be kept at the secondary queue Q_2 to wait for a chance to run while their deadlines are not yet missed. The primary queue Q_1 is never empty as long as the secondary still has requests. This is, because, requests in the secondary queue Q_2 are moved back to the primary

queue as soon as one or more slots are available. New incoming requests still have a chance to be directed to the primary queue.

ADUs in the primary queue have a higher probability of meeting their response time requirements. Therefore, they have lesser risk of deadline miss. Scheduling for this queue would be chosen to best support the secondary queue. The secondary queue however always uses LLS, since data units are more likely to miss their deadlines. After the burst occurs, the data units stored in Q_2 are pushed back to Q_1 and processed in a timely manner.

4 Performance Evaluation

4.1 Experimental Setup

We have implemented our approach as a scheduler component in our Synergy distributed stream processing system [6] and evaluated its performance. Synergy runs on top of the FreePastry library [8], an open source implementation of the Pastry DHT which is used for component discovery and collecting statistics. Our system is deployed on a 10/100 LAN network of Debian Linux 2.6.20 workstations consisting of Intel Pentium 4 2.66GHz and Intel Xeon 3.06GHz processors, whose main memory varied from 1GB to 2GB of RAM. Our system is written in Java and was developed with Eclipse using Java 1.6.0. We used the timing function provided by the JVM 1.6.0 with time granularity of $1msec$, which is adequate for our experiments. We run a series of experiments to evaluate the performance and demonstrate the working of our approach.

4.2 Experimental Results

In the first set of experiments our goal was to evaluate the performance of our approach under different burst intensities. To model the fluctuating nature of bursts, we designed a burst pattern that has two bursts with a period of normal rate after each burst. Our experimental system employs 6 unique services instantiated as unique components on the processing nodes. Each component is replicated and available at multiple nodes. Each experiment instantiates concurrently 11 applications. Each application submits a unique service request structured as a DAG; each service request invokes 4 to 6 service components which are located on different nodes. To generate the service request, the source component of an application dispatches a stream of data units which is 250 bytes long. The normal rate required by each service request ranges from 8 Kpbs to 45 Kbps, which is, 80% of the maximum request service rate, as can be obtained through profiling. We run a series of experiments with different burst intensities from no bursts 0% (normal rate) to 100% burst. Results are averaged over 5 runs with 90% confident interval where possible.

To demonstrate the working and benefits of our approach we have implemented two additional scheduling approaches as follows: (a) **First Come First Served (FCFS)**: orders the data units on the scheduler's queue based on the order in which they arrive at the node, (b) **Earliest Deadline First (EDF)**:

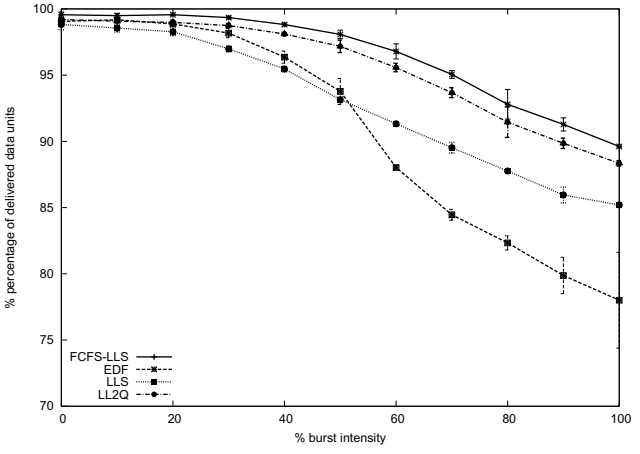

Fig. 2. Percentage of delivered data units on time, under various burst intensities

uses the deadline of the data units to decide the ordering; the data unit with the smallest deadline is ordered first. To demonstrate the advantages of our two-queue approach, we have compared the following strategies: **FCFS-LLS** uses the FCFS scheduler for the primary queue and the LLS scheduler for the secondary queue, while the **LLS-LLS (LL2Q)** strategy uses the LLS scheduler for both queues. We have also compared our two-queue scheduling approach with an approach that uses a single queue for scheduling. In particular, the **LLS** approach uses a single queue where the data units are ordered based on the least laxity scheduling algorithm, while the **EDF** scheduling approach uses a single queue where the data units are ordered based on the EDF scheduling.

On-time Delivery: In this experiment we measure the percentage of data units that are delivered on time as a function of burst intensity. Our approach that employs two-queues for scheduling leads to over 90% of data units that are delivered, as shown in figure 2. As the figure indicates, especially the FCFS-LLS scheduling has the highest percentage. An interesting observation that we notice is that the two-queue approach LLS-LLS performs better than single queue LLS. The reason for this is that the single queue LLS experiences a large number of context switching by which data units are frequently switched in the queue for the incoming data units which have smaller laxities. The longer the single queue, the worse the problem is, and as a result this significantly affects the system performance. In the two-queue approach, the problem is less since the primary queue's size is smaller, just enough to keep requests that their response time are guaranteed. When FCFS is used as the primary queue's order, there is no problem of context switching.

Dropped Data Units: The percentage of dropped data units as the results of deadline missing is shown in figure 3. As the figure indicates, both two-queue scheduling approaches perform better than any single-queue approach, especially with higher burst intensities.

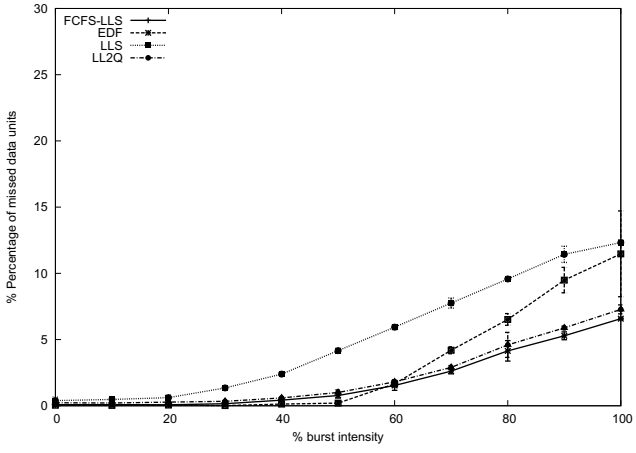

Fig. 3. Percentage of data units with missed deadlines, under various burst intensities

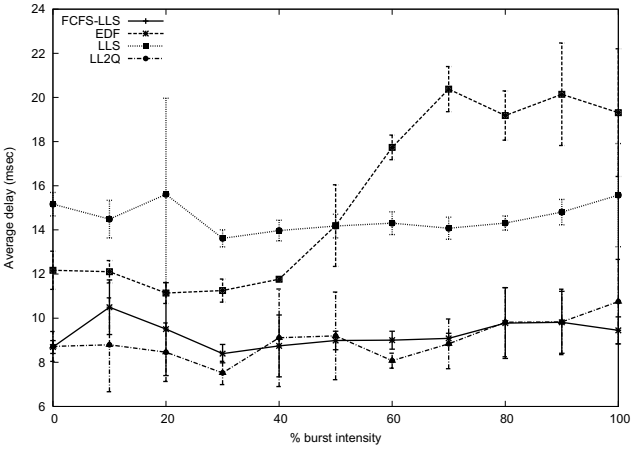

Fig. 4. Average end-to-end delay, in msec, under various burst intensities

Average End-to-End Delay: In the next experiment we measure the end-to-end delay experienced by the data units in the system. Figure 4 shows the average end-to-end delay of data units with different scheduling approaches. Both two-queue scheduling approaches decrease the average delay and are not affected by the burst intensity. With LLS the data units with the least laxity are given top priority. If they are deliverable, it means that their laxities are still positive and their delays are small accordingly. The two-queue approach with LLS even enhances the delay. Since data units are in the primary queue, with small size and given a top priority, will not wait long for processing. When they are processed, their delays are still small. This is one of interesting features of the two-queue approach.

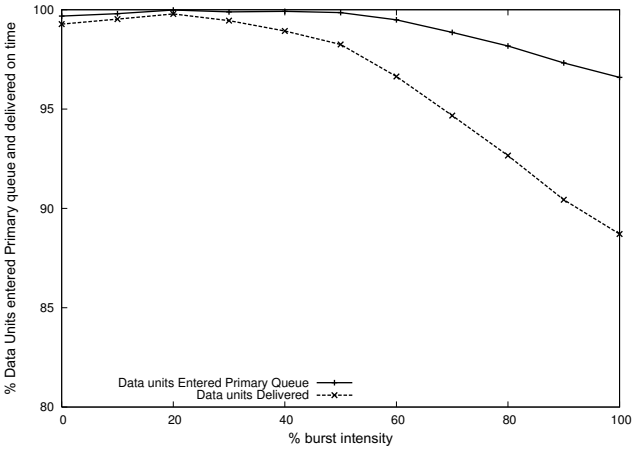

Fig. 5. Percentage of data units inserted and delivered on-time in the primary queue

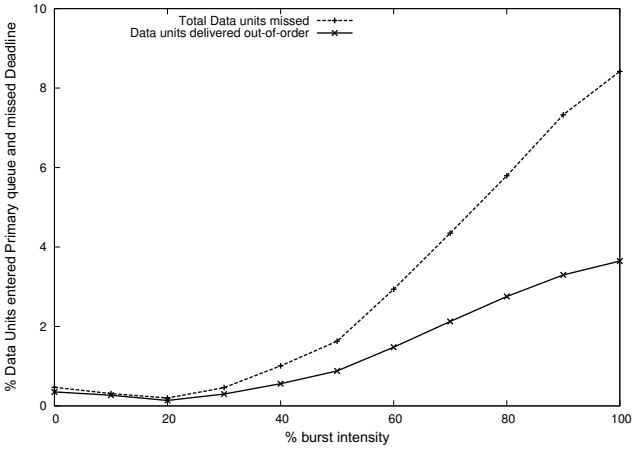

Fig. 6. Percentage of data units of the primary queue with missed deadlines and the corresponding percentage of data units delivered out of order

Performance of the two queues approach: In this set of experiments we illustrate the working of our approach by showing separately the behavior of each of the queues. Figure 5 shows the percentage of the data units of the primary queue that enter the primary queue and those that are delivered on-time. Note that there is some percentage of data units of the primary queue that are not delivered on time. As the burst intensity increases the percentage of data unit delivery drops. There are two reasons for this: As figure 6 shows, the out-of-order delivery accounts for about 50% of the misses. Furthermore, some data units that enter the primary queue and are then delivered, they could come from the secondary queues in the upstream nodes. The problem of out-of-order delivery could be easily mitigated by post-ordering at the destination based on

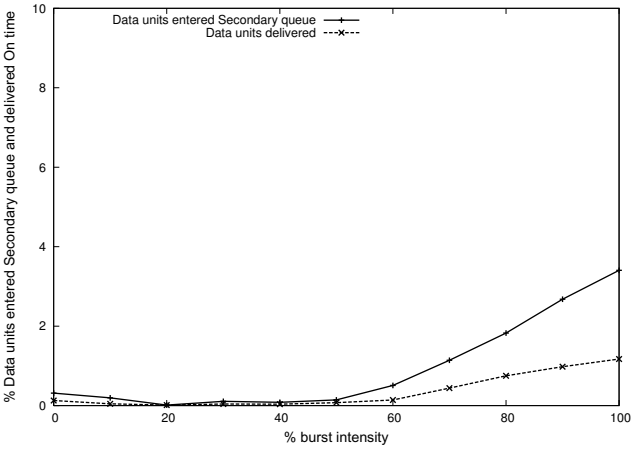

Fig. 7. Percentage of data units inserted and delivered on-time in the secondary queue

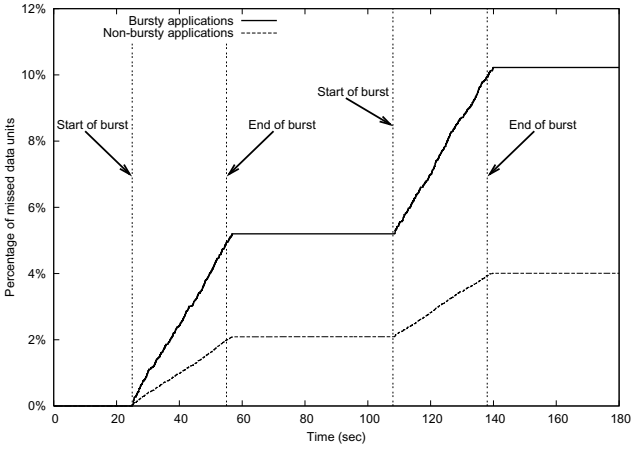

Fig. 8. EDF: Percentage of missed data units for bursty and non-bursty applications

their ID-numbers. Figure 7 shows about 4% of workload are diverted to the secondary queue, and half of this load are on-time deliverable.

Mitigating effects of bursty applications: In the next set of experiments we explored how the approach handles a mix of bursty and non-bursty applications. Using the same double peak burst pattern as in the first experiment, however, 9 applications, accounted for 70% of total workloads, have 100% burst intensity while the remaining applications, or 30% of total workloads have no bursts. This set of experiments demonstrates how our system handles both bursty and non-bursty applications.

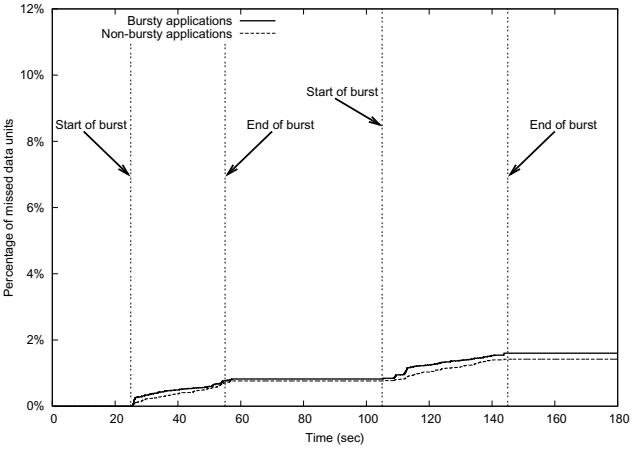

Fig. 9. FCFS-LLS: Percentage of missed data units for bursty and non-bursty applications

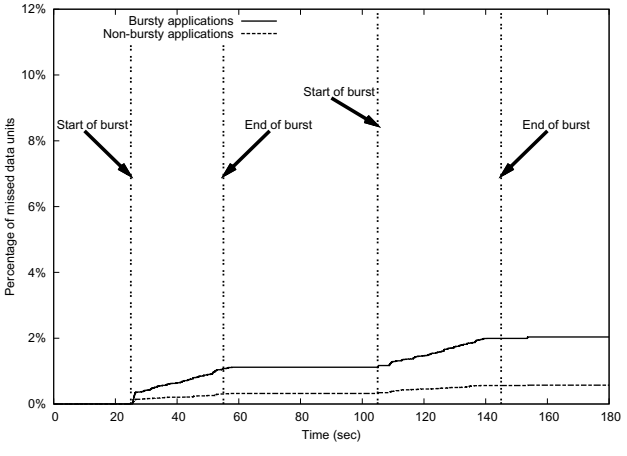

Fig. 10. LL2Q: Percentage of missed data units for bursty and non-bursty applications

Figures 8, 9, 10, and 11 show the percentage of data unit misses of bursty and non-bursty applications on different scheduling approaches as a function of time. Figure 8 and 11 show non-bursty applications with single queue scheduling, both EDF and LSS, are affected by the burst. This results into an increased number of data unit misses as soon as the bursts occurs. Figure 9 and 10, on the other hand, show data unit misses are low with FCFS-LLS and LL2Q (LLS-LLS) scheduling. With an appropriate size, the primary queue does not give bursty applications the chance to greedily occupy the processing queue due to their high rate. If the queue is large, as in a single queue scheduling, the situation will get worse with bursts. Data units from bursts application will overwhelm the queue and

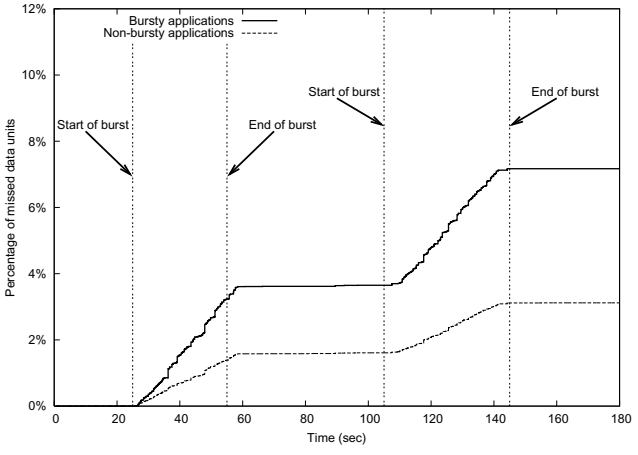

Fig. 11. LLS: Percentage of missed data units for bursty and non-bursty applications

data units of none-burst application will be affected by a high miss rate together with burst application since the system is unable to handle. This observation is significant for critical application that need to maintain a required throughput and not affected by bursts.

5 Related Work

Distributed stream processing systems have become increasingly popular in recent years for the development of applications that are characterized by high-rate, large-volume data streams. Examples include Aurora [2], STREAM [3], TelegraphCQ [1] and Cougar [4].

We have developed Synergy, a distributed stream processing system in [6] and have investigated various challenges related to dynamic rate allocation in [17], decentralized media streaming and transcoding [10]. Following are recent works (including ours) that propose solutions to accommodate bursts.

In [11], the authors propose a two-tie distributed control algorithm with the goal of maximizing the entire system weighed throughput, achieve low end-to-end latency, and stabilize the system coping with bursty workload. In the first tie, Lagrange multipliers are used to maximize the resource utilization. In the second tie, CPU and flow control algorithms are used to adjust input rates with feedback from downstream components. With this mechanism, any exceed input rate will be dropped. In our approach, in contrast, the exceed input rates instead of being dropped, are diverted to a secondary queue and re-scheduled to be processed later with best effort. To avoid context switching overhead and decrease memory cache miss, in [11] a batch scheduling is employed to process several data units at a time, however, this scheduling strategy does not address bursts.

In [12], the authors propose centralized and distributed load shredding approach to address bursts in distributed stream processing systems. A number of shredding plans are generated in advance for certain load conditions, so that the system can react to overload fast and in a lightweight manner. Under this approach, any input rates that exceed the maximum allowed rate in pre-configured shredding plans will be dropped. Our approach, with two-queue scheduling, is able to claim back data units that would otherwise be dropped, like in this approach.

In [13], the authors propose a multi-parametric programming approach to maximize the system utility in response to changing workloads. The approach consists of an off-line and an on-line version. The off-line version transforms the utility optimization problem into a linear function of CPU utilization, while the on-line version produces optimal solutions based on workload variation in polynomial time. This approach also shreds input rates that exceed the rate establised by their solution.

Our previous work on burst management [14] presents a solution that accommodates bursts by re-distributing the load among the processing nodes of the system. Furthermore, our previous work does not differentiate between the applications, and treats all of them the same. In this paper we present a totally different approach to accommodate bursts using a two-queue scheduling approach. In particular, this paper focuses on maximizing the potential of each individual application (and of the system overall). The method we present considers the urgencies of the applications and their resource requirements to decide the scheduling ordering of the data streams on the system resources. This maximizes the probability that the deadlines of the most urgent applications (and thus most important applications) will be met.

In [5], the authors propose two queues to handle bursty workload for storage systems. Our work, on the other hand, studies the effect of employing a secondary queue for distributed stream processing systems which, to the best of our knowledge, has not been proposed before. This is a much more difficult problem because: (1) stream processing applications are distributed and thus the occurrence of a burst is not confined in one node, rather it can affect remote nodes that run components of the same applications, and (2) distributed stream processing applications have end-to-end deadlines which are affected by all nodes running components of the applications. Our approach aims to meet the end-to-end deadlines of the applications. Furthermore, in order to deal with the unique requirements of the distributed stream processing systems, we have made additional contributions: we have implemented several different scheduling algorithms and we have studied the behavior of the two-queue structure under these different scheduling strategies. In particular, we have implemented and compared the following scheduling algorithms: least laxity scheduling, earliest deadline first and first come first served.

6 Conclusions

In this paper, we have investigated the problem of accommodating unpredicted data bursts in streaming applications deployed over cloud computing infrastructures. Our approach takes into account the fluctuation nature of bursts and their effects on existing applications, and employs two scheduling queues with different scheduling policies. When bursts occur, any excess workload is diverted to a secondary queue, where requests can be processed whenever the node has any unused processing power. Our experimental results on our Synergy distributed stream processing system show that our two-queue approach efficiently reduces data unit drops, improves end-to-end delay compared to other popular single queue approaches, and significantly mitigates the effects of bursts on non-bursty applications.

Acknowledgements. This research has been supported by the European Union through the Marie-Curie RTD (IRG-231038) Project and by AUEB through a PEVE2 Project.

References

1. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: CIDR, Asilomar, CA (January 2003)
2. Tatbul, N., Çetintemel, U., Zdonik, S.B., Cherniack, M., Stonebraker, M.: Load Shedding in a Data Stream Manager. In: VLDB 2003, Berlin, Germany, pp. 309–320 (2003)
3. Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., Widom, J.: STREAM: The Stanford Data Stream Management System (March 2005)
4. Madden, S., Gehrke, J.: Query Processing in Sensor Networks. IEEE Pervasive Computing, vol 3(1) (March 2004)
5. Lu, L., Varman, P., Doshi, K.: Graduated QoS by Decomposing Bursts: Don't Let the Tail Wag Your Server. In: ICDCS 2009, Montreal, QC, Canada, pp. 12–21 (June 2009)
6. Repantis, T., Gu, X., Kalogeraki, V.: Synergy: Sharing-Aware Component Composition for Distributed Stream Processing Systems. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 322–341. Springer, Heidelberg (2006)
7. Kalogeraki, V., Melliar-Smith, P.M., Moser, L.E.: Dynamic Scheduling of Distributed Method Invocations. In: IEEE Real-Time Systems Symposium (RTSS), Orlando, FL (December 2000)
8. FreePastry (2006), `http://freepastry.org/FreePastry`
9. Drougas, Y., Kalogeraki, V.: RASC: Dynamic Rate Allocation for Distributed Stream Processing Applications. In: International Parallel and Distributed Processing Symposium (IPDPS), Long Beach, CA (March 2007)
10. Chen, F., Kalogeraki, V.: RUBEN: A Technique for Scheduling Multimedia Applications in Overlay Networks. In: Globecom 2004, Dalas, TX (November 2004)

11. Amini, L., Jain, N., Sehgal, A., Silber, J., Verscheure, O.: Adaptive Control of Extreme-scale Stream Processing Systems. In: ICDCS 2006, Lisboa, Portugal (2006)
12. Tatbul, N., Çetintemel, U., Zdonik, S.: Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In: VLDB 2007, Vienna, Austria, pp. 159–170 (September 2007)
13. Chen, Y., Lu, C., Koutsoukos, X.: Optimal Discrete Rate Adaptation for Distributed Real-Time Systems. In: Real Time Systems Symposium (RTSS), Tucson, AZ (December 2007)
14. Drougas, Y., Kalogeraki, V.: Accommodating Bursts in Distributed Stream Processing Systems. In: 23rd International Parallel and Distributed Processing Symposium (IPDPS), Rome, Italy (May 2009)
15. Amazon Elastic Computer Cloud (Amazon EC2), `http://aws.amazon.com/ec2/`
16. IBM Cloud Computing, `http://www.ibm.com/ibm/cloud/`
17. Drougas, Y., Kalogeraki, V.: RASC: Dynamic Rate Allocation for Distributed Stream Processing Applications. In: International Parallel and Distributed Processing Symposium (IPDPS), Long Beach, CA (March 2007)