# Hot-Spot Prediction and Alleviation in Distributed Stream Processing Applications

Thomas Repantis Vana Kalogeraki
Department of Computer Science and Engineering
University of California, Riverside, CA 92521
{trep,vana}@cs.ucr.edu

## Abstract

*Many emerging distributed applications require the real-time processing of large amounts of data that are being updated continuously. Distributed stream processing systems offer a scalable and efficient means of in-network processing of such data streams. However, the large scale and the distributed nature of such systems, as well as the fluctuation of their load render it difficult to ensure that distributed stream processing applications meet their Quality of Service demands. We describe a decentralized framework for proactively predicting and alleviating hot-spots in distributed stream processing applications in real-time. We base our hot-spot prediction techniques on statistical forecasting methods, while for hot-spot alleviation we employ a non-disruptive component migration protocol. The experimental evaluation of our techniques, implemented in our Synergy distributed stream processing middleware over PlanetLab, using a real stream processing application operating on real streaming data, demonstrates high prediction accuracy and substantial performance benefits.*

## 1. Introduction

A variety of emerging applications require real-time processing of high-volume, high-rate data that are updated continuously. Examples include analyzing the input provided by website visitors to provide relevant advertising, monitoring network traffic to detect intrusions or update router configuration, customizing news feeds to user interests, or processing financial trading data for recommendations or alerts. This type of applications has given rise to a new class of systems, called *Distributed Stream Processing Systems (DSPSs)* [1, 8, 10, 13, 14]. In DSPSs, reusable components, located on geographically distributed nodes, process continuous data streams in real-time. These components are composed dynamically to form distributed applications.

Distributed stream processing applications have Quality of Service (QoS) requirements, expressed in terms such as end-to-end delay, throughput, or miss rate. For example, an alert needs to be raised within a certain time frame after an intrusion, or a trading recommendation needs to be made while processing financial data at certain rates. Adhering to such QoS requirements is crucial for the dependable operation of a DSPS. The first step towards satisfying the QoS requirements of stream processing application is taking them into account during the application composition [8, 14]. However, as the incoming data rates may increase at run-time, due to external events such as a network attack or a rapid popularity growth of some news event, an application execution may cease to adhere to the requested QoS. In fact, the distinct characteristic of stream processing applications is that the data to be processed arrive in high rates, and often in bursts [24, 25]. Under such dynamically changing conditions, providing application QoS is a challenging task. The problem is complicated further by the large scale and the distributed nature of a DSPS. Accurate centralized decisions are infeasible, due to the fact that the global state of a large-scale DSPS is changing much faster than it can be communicated to a single host.

In this paper we study the problem of predicting and alleviating application hot-spots in a DSPS. Current approaches for addressing load fluctuations in DSPSs [3, 18, 24, 25], including our previous work [15], focus on avoiding or resolving hot-spots in the system resources, in other words overloaded nodes. We refer to this kind of hot-spot detection and alleviation as node-oriented. The focus of this work, on the other hand, is on detecting and alleviating hot-spots in the application execution, in other words applications that persistently fail to meet the QoS required by the user. We call this kind of hot-spot detection and alleviation *application-oriented*. We believe that application-oriented hot-spot detection and alleviation are as important as their node-oriented counterparts for the following key reasons: i) Application-oriented hot-spot detection is more sensitive and can be triggered even when a node is

underloaded. Even when running on a moderately loaded node, an application may not be meeting its QoS requirements (e.g., if they are stringent), thus experiencing a hot-spot. On the other hand, with node-oriented hot-spot detection, by the time a node is overloaded many of the applications using that node will already have violated their QoS requirements. ii) Application-oriented hot-spot alleviation allows more fine-grained hot-spot alleviation. Depending on the individual applications' QoS demands, only some instead of all the applications that are using a node may be suffering, and thus only these applications may need to be migrated. On the contrary, node-oriented hot-spot alleviation aims at reducing a node's load, irrespective of which of the applications experience overload. iii) Most importantly, application-oriented hot-spot detection enables taking *proactive* measures with regards to application performance, to prevent severe degradation of application QoS.

Specifically, this paper makes the three following main contributions: i) We propose a framework built on statistical forecasting methods, to accurately predict QoS violations at run-time and proactively identify application hot-spots. In order to achieve this, our prediction framework binds workload forecasting with execution time forecasting. To accomplish workload forecasting we predict rate fluctuations, exploiting auto-correlation in the rate of each component, and cross-correlation between the rates of different components of a distributed application. To accomplish execution time forecasting we use linear regression, an established statistical method, to accurately model the relationship of the application execution time and the entire workload of a node, while dynamically adapting to workload fluctuations. ii) To react to predicted QoS violations and alleviate hot-spots we enable nodes to autonomously migrate the execution of stream processing components using a non-disruptive migration protocol. Candidate selection for migration is based on preserving QoS. We employ prediction again to ensure that migration decisions do not result to QoS violations of other executing applications. To drive migration decisions in a decentralized manner we build a load monitoring architecture on top of a Distributed Hash Table (DHT) [16]. iii) We have implemented our techniques in Synergy [14], our distributed stream processing middleware[1]. To validate our approach we have deployed our middleware on the PlanetLab [5] wide-area network testbed and we have run experiments of a real network monitoring application [20] operating on traces of real TCP traffic [22]. Our experimental evaluation demonstrates high prediction accuracy, with an average prediction error of 3.7016%, and substantial benefits in application QoS, achieved by migrations that are completed in approximately 1s.

[1]Synergy is implemented as a multi-threaded system of about 35,000 lines of Java code and more information is available at http://synergy.cs.ucr.edu/

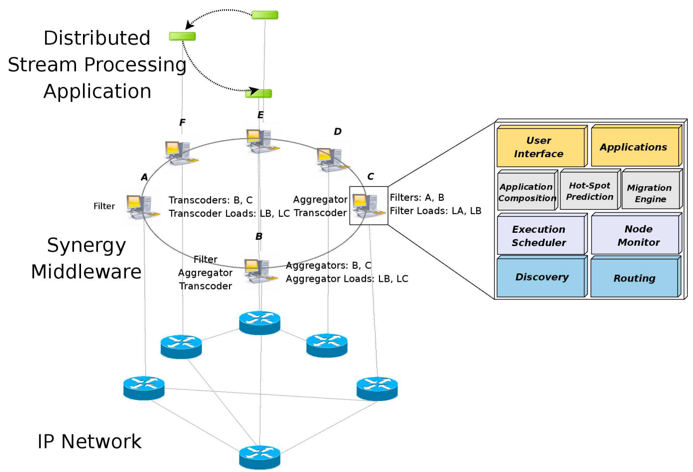


**Figure 1. The basic blocks of Synergy.**

## 2. The Synergy Middleware

In this section we present a brief overview of our Synergy distributed stream processing middleware. Synergy is a middleware designed to provide QoS support for distributed stream processing applications. In Synergy, data streams, consisting of independent data tuples, arrive continuously from external sources (such as web users, monitoring devices, or a sensor network) and need to be processed by stream processing components in real-time. Each component is a self-contained software module, that offers a predefined operator. The operators can be as simple as a filter or a join, or as complex as transcoding or encryption. Components are deployed in the distributed nodes of the Synergy middleware according to their individual software capabilities or following criteria for the optimization of the performance of the whole system [1, 13].

The nodes of our distributed stream processing middleware are connected via overlay links on top of the existing IP network. The application component graph is built on top of the middleware, as shown in Figure 1. The basic blocks of the Synergy middleware running on each node are shown in Figure 1. Synergy offers several benefits: i) It enables efficient component composition that meets end-to-end QoS demands by sharing resources, components, and streams [14]. ii) It provides a low overhead resource monitoring facility. iii) It allows fast stream and component discovery by utilizing the underlying DHT infrastructure [16].

The user executes a distributed stream processing application by submitting a request at one of the nodes of the middleware, specifying the required operators and their dependencies. Then, the system runs a composition algorithm to select the components on the nodes to accomplish the application execution. These will constitute the application component graph, that represents the sequence of compo-

nent execution and the corresponding hosting nodes.

We have extended Synergy's architecture to enable decentralized load monitoring [15], built on top of the DHT we use for component discovery. We have implemented a distributed inverted index on top of the DHT. This way we associate operator names with handlers to nodes hosting components offering these operators, together with the current load values of these nodes. For example, in Figure 1 on the left of each node are listed the components this node is offering, while on the right of each node are listed the handlers and loads this node is responsible for maintaining. Node B is responsible for keeping the handlers for the components that offer an aggregator operator. Therefore it keeps the handlers of nodes B and C, as well as the loads of nodes B and C. Whenever a node's load changes, it consults the DHT to determine the nodes responsible for holding the handlers for all the components it offers. It then sends load update messages to them. For example, in Figure 1 node B that offers a filter, an aggregator, and a transcoder, will send its load update messages to the responsible nodes, C, B, and A, respectively. To avoid the communication overhead caused by updating, we enable the nodes to inform the monitoring nodes only when a significant change in their load occurs. Configuration changes such as node arrivals, departures, failures, or balancing of operator keys among nodes are handled by the DHT [16].

We use our decentralized load monitoring architecture to cope with application hot-spots. We define an application hot-spot as a node in the application component graph in which the application execution persistently fails to meet the QoS required by the user. The end-to-end QoS requirements, which are specified when requesting an application, may among others include end-to-end execution time, throughput, or miss rate. Although our schemes are generic to additive QoS metrics linearly related to rate, we focus on the end-to-end execution time metric denoted by $q_t$.

# 3. Application Hot-Spot Prediction

The goal of proactive application hot-spot detection is to predict end-to-end execution time QoS violations. In order to achieve this goal we employ: i) Computation of the application "slack time" $t_s$ (Section 3.1), to determine the maximum local execution time allowed by the application QoS, before missing its end-to-end execution time requirement. ii) Local execution time prediction based on an application's incoming rate and using linear regression, to determine whether the maximum local execution time will be reached or exceeded (Section 3.2). iii) Rate prediction based on auto- and cross-correlation between stream processing components, to determine the future workload that defines the future execution time (Section 3.3).

## 3.1 End-to-End to Local Execution Time Translation

We predict an application hot-spot by examining the "slack time" of the application on every component of the application component graph. The slack time represents how close we are to violating the end-to-end execution time requirement of the application. Let $q_t$ represent the end-to-end execution time requirement of the application. $q_t$ includes the execution and communication times spent for a tuple to traverse the entire application component graph. Thus, we define the slack time $t_s$ of an application as the difference between the required end-to-end execution time $q_t$ and the predicted end-to-end execution time. As the application executes, its slack time is computed for every tuple, on every component of the application component graph, based on the local prediction of the end-to-end execution time. The predicted end-to-end execution time includes the execution and communication times spent for a tuple to reach the current component, $t_e$ and $t_c$ respectively, the predicted execution times $\hat{t_e}$ needed for the current and its downstream components to process the data tuple, as well as estimated average communication times $\bar{t_c}$ needed for the data tuple to traverse the rest of the application component graph. For example, in Figure 3 the predicted end-to-end execution time as it is calculated in component B is the sum of $t_{e(A)}$, $t_{c(A \rightarrow B)}$, $t_{e\hat{(B)}}$, $t_{c(B \bar{\rightarrow} D)}$, and $t_{e\hat{(D)}}$. In order to avoid a QoS violation, the predicted end-to-end execution time needs to be less than the required end-to-end execution time $q_t$, in other words, the slack time $t_s$ needs to be positive, for every component $i$ of the $v$ components of the application component graph:

$$t_{s(i)} = q_t - ( \sum_{j \in 1 \dots i-1} t_{c(j \rightarrow j+1)} + \sum_{j \in 1 \dots i-1} t_{e(j)} + \sum_{j \in i \dots v-1} t_{c(j \bar{\rightarrow} j+1)} + \sum_{j \in i \dots v} t_{e\hat{(j)}} ) > 0 \quad (1)$$

The above single-path computation will identify a hot-spot in the path where it exists. For example, if in Figure 3 component C is overloaded, the path $A \rightarrow B \rightarrow D$ will not detect a hot-spot, while path $A \rightarrow C \rightarrow D$ will. In order for the above hot-spot prediction to take place, the estimated average communication times, and the predicted execution times must be computed. The estimates for the communication times are available from the application composition phase [14] and can be updated periodically. The predicted execution times are derived locally on every node hosting a component of the application component graph, as explained in the following Section 3.2. They are then propagated to all nodes participating in the application execution using a feedback loop passing through the source. The feedback loop allows us to piggyback the predicted execution times on the data tuples, to minimize the communication

overhead. For example, in the application component graph shown in Figure 3 when the node hosting component D calculates the component's next predicted execution time for this application, it propagates it to the node hosting component A, which forwards it to the nodes hosting components B and C. Similarly, the rest of the nodes propagate their predicted execution times. Using the predicted execution times to compute the slack time on every component enables us to predict locally whether the end-to-end execution time requirement of the application will be violated.

### 3.2 Local Execution Time Prediction

In this section we explain how we predict the local execution time $\hat{t_e}$ needed to process a data tuple of an application. The prediction takes place at each node hosting a component of the application. $\hat{t_e}$ is used to compute the next slack time $t_s$ of the application using Equation 1. The local execution time for a data tuple (the time elapsed between the arrival and the departure of the tuple) is the sum of the processing time to process the tuple, and the queueing time the tuple has to wait in the scheduler's queue while other tuples are being processed. While the processing time is constant for a given tuple size, the queueing time depends on the load of the processing node, in other words on the rates (incoming tuples to be processed per time unit) and processing times of the applications currently being executed on the node. Using queueing theory, one can derive average values for the queueing time, assuming an M/M/1 queueing model [14], or a more general M/G/1 model that makes no assumptions regarding the service rate, in which case the queueing time is given by the Pollaczek-Khinchin mean value formula [9]. However, we chose not to predict the execution time using queueing theory for the following reason: The arrivals of data tuples may not always be accurately approximated with a Poisson distribution if rate fluctuations or bursts occur. Such rate variations are quite common in distributed stream processing applications [24]. Accurate prediction during such fluctuations is however crucial. We use linear regression to predict the execution time of an application [12]. Since data tuples arrive in high rates, prediction is more fine-grained than node load changes.

To predict the local execution time $\hat{t_e}$ of an application using a component on a node, we need to derive the relationship between $\hat{t_e}$ and the total rates $r_t = \sum_{l \in 1 \ldots a} r_l$ of all $a$ applications currently using components on that node. While for increasing $r_t$ one expects $\hat{t_e}$ to increase, the trend of the increase is not clear without making any assumptions regarding the arrival pattern of the data tuples. We approximate the relationship using linear regression and our experimental results show good fitting for increasing rates. Figures 14, 15 show the relationships between the execution times of different components of a stream processing application and the rates of the applications currently running on the nodes hosting them, obtained from our implementation over Planetlab. Linear relationship of execution time and rate is also consistent with earlier works [23, 24].

Each node maintains a series of $(t_e, r_t)$ pairs, for each application a component of which the node is hosting. The series is maintained as a sliding window of the $k$ most recent values. The execution time is measured every time a data tuple for an application is processed, while the total rate is measured as the sum of rates of all applications, data tuples of which were processed since the last time a data tuple of that application was processed. If the rate of any application increases, it affects the execution time of other applications on the same node due to queueing delays. We estimate the conditional expected value of $t_e$, given a predicted value for $r_t$. We use linear regression, and assuming we have $k$ pairs so far, the linear function is $t_e = a + b \cdot r_t$ and the least square estimators $a$ and $b$ are:

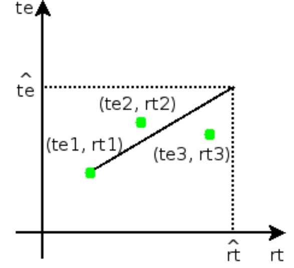


**Figure 2. Linear regression.**

$$a = \bar{t_e} - b \cdot \bar{r_t} \qquad b = \frac{\sum_{j \in 1 \ldots k} (r_{t(j)} - \bar{r_t}) \cdot (t_{e(j)} - \bar{t_e})}{\sum_{j \in 1 \ldots k} (r_{t(j)} - \bar{r_t})^2} \quad (2)$$

where the average values $\bar{t_e}$ and $\bar{r_t}$ are:

$$\bar{t_e} = \frac{\sum_{j \in 1 \ldots k} t_{e(j)}}{k} \qquad \bar{r_t} = \frac{\sum_{j \in 1 \ldots k} r_{t(j)}}{k} \quad (3)$$

In order to enable proactive hot-spot detection, we base the prediction of the execution time $\hat{t_e}$ of an application on the predicted rates of the applications running on components of the node, $\hat{r_t} = \sum_{l \in 1 \ldots a} \hat{r_l}$. (We explain how $\hat{r_l}$ for an application $l$ is derived in the following Section 3.3.) Assuming an estimated value for the next $\hat{r_t}$, we predict $\hat{t_e}$ using the above equations. Specifically, as shown in Figure 2, we use the $k$ pairs of $(t_e, r_t)$ values to calculate $a$ and $b$ and then given an estimated $\hat{r_t}$ we predict $\hat{t_e}$ using the following formula:

$$\hat{t_e} = a + b \cdot \hat{r_t} \quad (4)$$

To evaluate the accuracy of our execution time prediction we calculate the estimated standard error of the slope $b$:

$$se(b) = \sqrt{\frac{\sum_{j \in 1 \ldots k} (t_{e(j)} - \bar{t_e})^2 - b \sum_{j \in 1 \ldots k} (r_{t(j)} - \bar{r_t})(t_{e(j)} - \bar{t_e})}{(k-2) \sum_{j \in 1 \ldots k} (r_{t(j)} - \bar{r_t})^2}} \quad (5)$$

If the estimated standard error $se(b)$ is above a heuristically set confidence level $C$, we do not employ execution time prediction. Instead we report the last measured application execution time value rather than a predicted future one. In general however the last measured value is not an accurate predictor, as it ignores the current rate.

### 3.3 Rate Prediction

In this section we describe how we predict the rate $\hat{r}$ of an application, which we use to calculate the sum of the rates of all applications running on components of a node, $\hat{r_t}$. The latter is used to predict the application execution time $\hat{t_e}$ using Equation 4. We base the prediction of the rate of every application that is using a component hosted on the node on both auto- and cross-correlation. We take into account auto-correlation by building our prediction of a component's future input rate on its previous input rate. This captures any self-similarity the application traffic may have, which has been known to be the case for various types of traffic in stream processing environments [24]. We take into account cross-correlation, by also building our prediction of the input rate of a component on the current input rate of a previous component in the application component graph. This captures the fact that preceding components observe changes in the application input rate before the current component. Since data flow from one component to the next, the observed trends are often seen in the current component as well. In particular, we identify the preceding component $m$ in the application component graph, the rate of which has the maximum correlation with the rate of the current component so far. In summary, we estimate the $k$-th input rate $\hat{r_k}$ of a component based on its previous input rate $r_{k-1}$, as well as the current and previous input rates of component $m$, $r_{k(m)}$ and $r_{k-1(m)}$ respectively.

We transfer the current input rate values to the downstream components using the same path followed by the data tuples, as shown in Figure 3. This way, for each of the previous $i$ components in the application component graph, a series of $(k-1)$ pairs $(r, r_{(i)})$ is built. This series associates the $(k-1)$ rate values $r$ of the current component with the $(k-1)$ rate values $r_{(i)}$ of each of the previous $i$ components. We use the Pearson Product Moment $R$, a popular correlation coefficient [12], to estimate how the rate of each of the previous $i$ components in the application component graph is correlated to the rate of the current component. We use the current ($k$-th) and previous ($(k-1)$-th) rates of the component $m$ with the maximum correlation coefficient, $\arg_m maxR_{(k)}$ and $\arg_m maxR_{(k-1)}$ respectively, as predictors for the rate of the current component. Hence, assuming we have $(k-1)$ pairs of recorded input rates so far, the estimated input rate for the current component is:

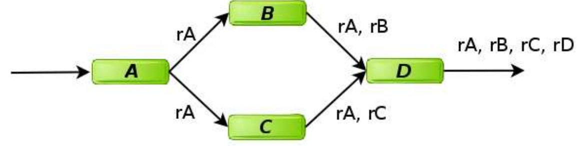


**Figure 3. Propagation of rate values for correlated rate estimation.**

$$\hat{r_k} = \frac{\arg_m maxR_{(k)}}{\arg_m maxR_{(k-1)}} \cdot r_{k-1} = \frac{r_{k(m)}}{r_{k-1(m)}} \cdot r_{k-1} \quad (6)$$

and the component $m$ is decided as the one with the maximum among all correlation coefficients $R_i$ of each preceding component $i$ in the application component graph:

$$R_i = \frac{\sum_{j \in 1\ldots(k-1)} (r_{j(i)} - \bar{r_{(i)}})(r_j - \bar{r})}{\sqrt{\sum_{j \in 1\ldots(k-1)} (r_{j(i)} - \bar{r_{(i)}})^2 \sum_{j \in 1\ldots(k-1)} (r_j - \bar{r})^2}} \quad (7)$$

where the average rate values of the $i$-th preceding and the current component, $\bar{r_{(i)}}$ and $\bar{r}$ respectively, are:

$$\bar{r_{(i)}} = \frac{\sum_{j \in 1\ldots(k-1)} r_{j(i)}}{k-1} \qquad \bar{r} = \frac{\sum_{j \in 1\ldots(k-1)} r_j}{k-1} \quad (8)$$

## 4. Application Hot-Spot Alleviation

### 4.1. Identifying the Components to Migrate

After an application hot-spot has been predicted, the next step is to determine which component execution(s) to migrate in order to resolve the hot-spot. We perform QoS projection and choose the migrations in such a way, so that the predicted execution times of the remaining applications in the node are within their QoS requirements.

Specifically, our goal is to determine the minimum number of migrations that will result to all the remaining applications satisfying their QoS requirements. In other words, we seek the minimum number of migrations that will reduce the sum of rates of all the applications in the node to such a degree, that all projected execution times for the remaining applications will be within their QoS requirements. More formally, and by building on the concepts introduced in Section 3, we migrate the component execution(s) that remove the minimum number of predicted rates $\hat{r}$ (from Equation 6), so that the predicted sum of application rates on the node $\hat{r_t}$ results to predicted execution times $\hat{t_e}$ (from Equation 4) such that, for every application remaining in the node, the slack time $t_s$ (from Equation 1) is positive.

This optimization problem lends itself to a dynamic programming solution in pseudo-polynomial time. After observing that usually one migration suffices to alleviate a hot-spot, and to minimize the execution time overhead, as migration decisions need to be taken online, we employ a simple heuristic of selecting for migration the component with the largest $\hat{r}$ until all slack times become positive.

### 4.2. Identifying the Target Nodes

Once a component the execution of which is to be migrated has been identified, the host to migrate to has to be decided. The choice for migration targets is made among the nodes that host the same component. Among them we try to identify a node probable to satisfy the migrating application's QoS requirements, while not violating the QoS of the applications currently running locally. Such nodes are most probable to be found among the ones that are predicted to be less loaded. Each node predicts its local load using linear regression, based on predicted rate values, using a methodology similar to the one described in Section 3.2. We use a simple model, according to which a component's load is proportional to the number of input data tuples it is receiving, which is an assumption also made by previous works [23, 24]. We store load information in a decentralized architecture [15] on top of the DHT, as was described in Section 2. By utilizing the load monitoring architecture a node determines the least loaded node offering the component the migration requires. After the migration target has been identified, the migration from the source to the target takes place, to resolve the application hot-spot.

To avoid QoS violations we perform QoS projection that predicts whether the QoS of the migrating and of the currently running applications will be able to be met after the migration has occurred. Once it has received a migration request, a node determines whether after accepting the migration it will be able to provide the migrating application its required QoS. Additionally, it determines whether the migration will not result to QoS violations for the locally executing applications. To achieve these goals, a migration target performs QoS projection involving the migrating and the currently running applications, that is similar to the one described in Section 4.1. Specifically, it ensures that by adding $\hat{r}$ for the new application, the sum of application rates on the node $\hat{r_t}$ will not result to a predicted $\hat{t_e}$ (from Equation 4) that results to a negative execution time slack for any application (from Equation 1). If that is the case, the migration is accepted and takes place using the migration protocol presented in [15]. Our current migration mechanism caters to stateless components and simple components whose state is captured in small buffers. State transfer is a separate issue by itself and worth future investigation.

## 5. Experimental Evaluation

To evaluate the performance of our hot-spot prediction and alleviation mechanisms we have implemented them in our Synergy distributed stream processing middleware and performed experiments over the PlanetLab [5] wide-area network testbed. We used 34 hosts, each one of them issuing a request for a distributed stream processing application. Each node was hosting stream processing components that were processing data tuples as they arrived. We set the application end-to-end delay QoS requirement to 20s.

To evaluate the accuracy of our prediction mechanisms we implemented a real stream processing application from the network traffic management domain, which we fed with real TCP traffic traces. We used a stream processing application from the Stream Query Repository [20], in which, assuming a packet capturing device installed in a network, a system administrator wishes to monitor the source-destination pairs in the top 5 percentile in terms of total traffic in the past 20 minutes over a backbone link. We generated the streaming data to be processed by replaying a TCP traffic trace available from the Internet Traffic Archive [22]. Similar results where obtained with the rest of the traces from [22]. The trace contained two hours' worth of all wide-area TCP traffic between the Lawrence Berkeley Laboratory and the rest of the world, consisting of 1.8 million packets. Each packet contained a timestamp, and fields defining the source and destination (IPs and ports), as well as the size of the packets exchanged between them. Our implementation of the above stream processing application to process the packet input over 20-minute windows to generate the monitoring output involved eight components, and screenshots are available at http://synergy.cs.ucr.edu/screenshots.html. Each node instantiated a different stream processing application that included all eight components of the application component graph, distributed randomly on different nodes of the system. Each node predicted the rate and the execution time of the components it was hosting using the statistical methods described in section 3. We plot predicted and actual values to show correlation and burstiness. The differences between actual and predicted values were also plotted but are omitted due to lack of space.

**Rate Prediction Accuracy.** In our first set of experiments we investigated the accuracy of our rate prediction algorithm described in Section 3.3. Figures 4, 5, 6, 7, and 8 compare the predicted rate for the individual components of an application to their actual rate. Similar results were obtained for all applications, as well as for the rest of the components of the application component graph, but are not included here due to lack of space. We observe that the predicted rate closely follows the measured rate for the different component types, namely sort, project, aggregate,

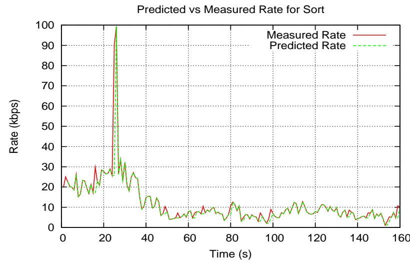


**Figure 4. Rate prediction accuracy for "sort".**

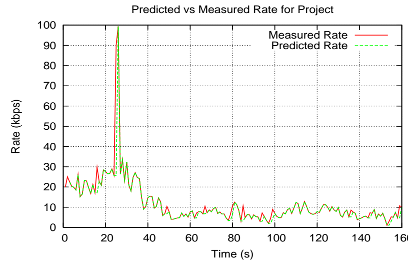


**Figure 5. Rate prediction accuracy for "project".**

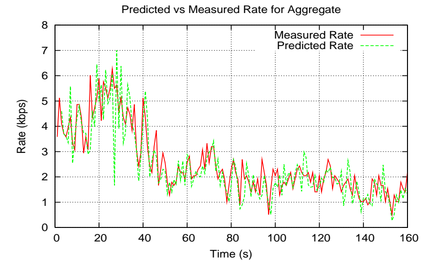


**Figure 6. Rate prediction accuracy for "aggregate".**

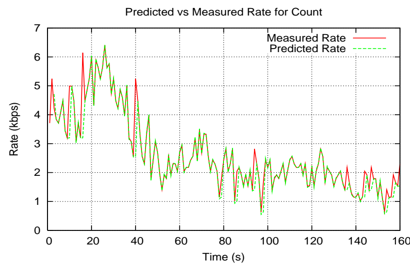


**Figure 7. Rate prediction accuracy for "count".**

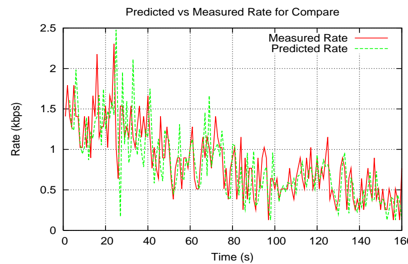


**Figure 8. Rate prediction accuracy for "compare".**

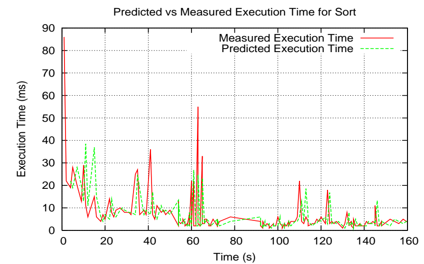


**Figure 9. "Sort" execution time prediction accuracy.**

count, and compare. Another interesting observation is the correlation in the rate between different components, for example between sort and project, or between aggregate and count. This indicates the significance of cross-correlation between different components in the application component graph, which we exploit in addition to auto-correlation to predict component rates.

**Execution Time Prediction Accuracy.** In our second set of experiments we investigated the accuracy of our execution time prediction algorithm described in Section 3.2. Figures 9, 10, 11, 12, and 13 compare the predicted to the actual execution time for the same set of components as in the rate prediction accuracy experiment. As was described in Section 3.2, the predictions are based on the sum of rates being processed by the node hosting each component. Note, that each component was hosted on a different node. The predicted execution time follows the execution time we measure. Cases where the prediction is very inaccurate are detected using the estimated standard error of the linear regression, as was described in section 3.2. This way, instead of the inaccurate predicted future execution time value the currently monitored value is reported.

**Execution Time Distribution.** In our third set of experiments we investigated the relationship between the execution time of the individual application components and the total rate for all applications being processed by each node hosting a component, shown in Figures 14, and 15 (similar Figures for the rest of the components are omitted due to lack of space). This enabled us to determine the accuracy of assuming a linear relationship between the two, which formed the basis of our linear regression-based execution time prediction algorithm described in Section 3.2. We observe that the relationship can be approximated by a line, excluding a few outliers. However this linear relationship is most evident when the total rate in the node is significant. If the node is lightly loaded, no significant queueing delays occur and therefore no significant variations in the execution time take place.

**Prediction Parameters.** In our fourth set of experiments we investigated various parameters regarding the prediction overhead and performance. In Figure 16 we show how rate prediction accuracy is affected when reducing the prediction frequency. Reducing the prediction frequency can enable the system to handle high rates, by avoiding the prediction overhead for every data tuple. We present the effect on prediction accuracy for the different components, when predicting the rate for every 1, every 50, every 75, and every 100 incoming data tuples. We observe that even by reducing the prediction overhead by a factor of 75, the prediction accuracy only drops by 8.775% on average, ranging from 2.0% for sort, to 14.375% for project.

Table 17 shows the average rate prediction error for the

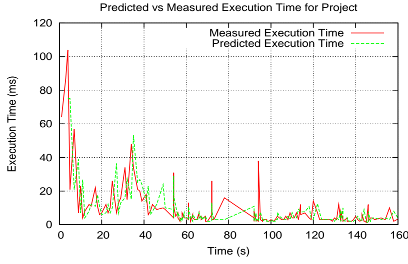


**Figure 10. "Project" execution time prediction accuracy.**

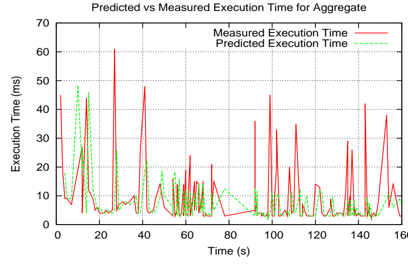


**Figure 11. "Aggregate" execution time prediction accuracy.**

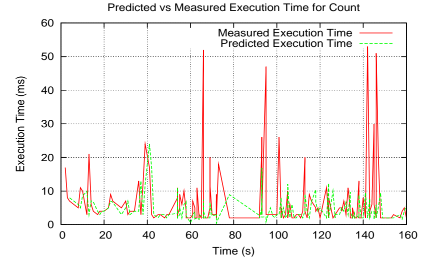


**Figure 12. "Count" execution time prediction accuracy.**

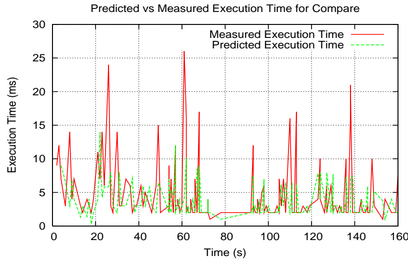


**Figure 13. "Compare" execution time prediction accuracy.**

**Figure 14. Execution time distribution for "sort".**

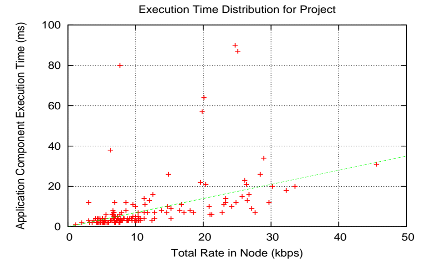


**Figure 15. Execution time distribution for "project".**

different application components. This provides a clear overview of the prediction accuracy. Even though some variation depending on the component semantics exists, the average prediction error is kept at 3.7016%. Table 18 shows the overhead in processing time for rate, execution time, and load prediction. The average overhead is 0.5984ms, which makes our algorithms suitable for online prediction.

**Application Performance.** In our fifth set of experiments we investigated the application benefits gained from our hot-spot prediction and alleviation mechanisms. Figure 19 shows the improvement in application QoS achieved by predicting application hot-spots and alleviating them using migration. The QoS metric displayed is the miss rate, defined as the number of data tuples that missed their QoS deadline, over the total number of data tuples that were produced by the source. The miss rate is displayed as a function of the system load. We inject additional load in the system by increasing the number of application component graphs each node requests from 1 to 10. When the system is underloaded not many application hot-spots occur and therefore their alleviation does not offer significant QoS advantages. However, as the system load increases, the miss rate increases drastically when hot-spots are not handled. Application hot-spot elimination controls this increase.

Figure 20 shows the benefit of hot-spot prediction and alleviation for the application performance. The performance metric displayed is the end-to-end application delay. Note that this delay is calculated only for the data tuples that did not miss their deadlines, as the ones that missed their deadlines are dropped by the local schedulers before reaching the receiver. While hot-spot prediction and alleviation enables the delivery of more data tuples as the load increases, it also maintains a lower average application end-to-end delay.

Figure 21 shows how a migration affects the performance of a particular application. For a load of 10 application requests per node, we show the end-to-end delay attained by delivered data tuples of one application. Approximately at data tuple #500 an application hot-spot occurs, resulting to an increase in the end-to-end delay. Our hot-spot elimination mechanism kicks in and decreases the end-to-end delay through migration approximately at data tuple #1200. It is also important to note that only the data tuples that were delivered within the application's QoS requirements are shown. As the application end-to-end delay increases, we can clearly observe a reduction in the number of delivered data tuples. After the hot-spot has been eliminated, the number of data tuples that miss their deadline decreases again and more points can be seen in the graph.

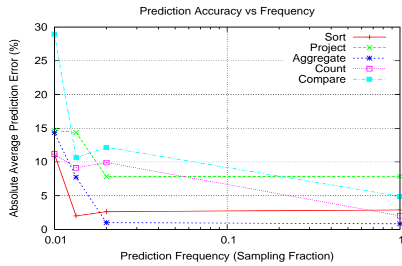


**Figure 16. Rate prediction accuracy versus prediction frequency.**

| Component | Average Prediction Error (%) |
|---|---|
| sort | 2.875 |
| project | 7.872 |
| aggregate | 0.838 |
| count | 2.019 |
| compare | 4.904 |

**Figure 17. Absolute average rate prediction error.**

| Component | Average Prediction Time (ms) |
|---|---|
| sort | 0.133 |
| project | 0.327 |
| aggregate | 0.509 |
| count | 0.836 |
| compare | 1.187 |

**Figure 18. Average total prediction time.**

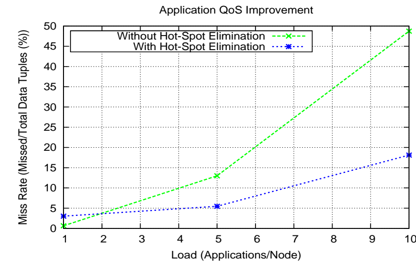


**Figure 19. Application QoS improvement.**

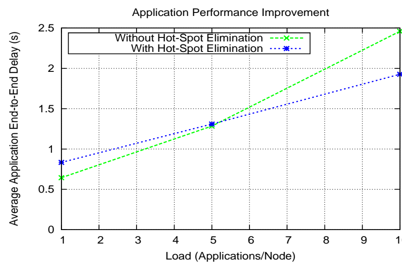


**Figure 20. Application performance improvement.**

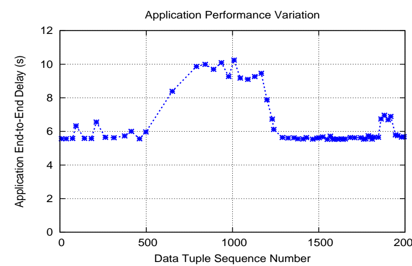


**Figure 21. Application performance variation.**

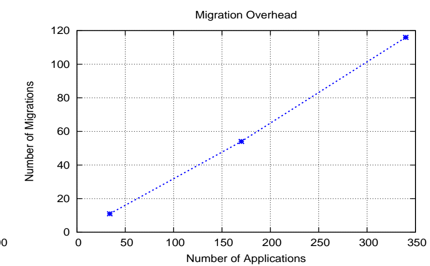


**Figure 22. Migration overhead.**

The Figure magnifies, focusing on 2000 of the total tuples.

In Figure 22 we show the migration overhead to achieve the hot-spot alleviation benefits. The number of migrations is shown as a function of the number of applications deployed in the system. We observe that the number of migrations grows linearly to the number of applications. On average, one migration every three applications is required. This shows that on average one every three applications experience a hot-spot at some point during the execution, which motivates the need for application-oriented hot-spot alleviation. This assumes that not many applications require more than one migration, in other words that the system is not so overloaded that a migration does not permanently resolve a hot-spot. We also measured the average time required to perform a migration to be 1144ms. This time included the complete distributed protocol execution described in Section 4.2. The short migration time, together with the fact that our migration protocol enables application execution to continue while the migration is taking place offline, make our hot-spot alleviation mechanism suitable for distributed stream processing applications with QoS demands. Prediction further facilitates fast reaction to a hot-spot, before massive QoS violations occur.

# 6. Related Work

Distributed stream processing systems have been the focus of a lot of recent research from different perspectives. Work on the placement of components to make efficient use of resources and to maximize application performance [1, 13] is complementary to ours. Any technique for deploying new components can be used, once all the nodes hosting a particular component type are overloaded. Additionally, the migration techniques presented in [13] can be used as an alternative to our migration protocol, complementing the prediction mechanisms for QoS violations presented here. Similarly, work on component composition [8, 14] or application adaptation [2, 6, 10] can assist in load balancing. Load balancing for distributed stream processing applications has also been studied [3,18,24,25]. We differ from these approaches in that we focus on the application QoS, rather than the system utilization. Furthermore, we propose a hot-spot prediction framework to drive proactive migration decisions. In our previous work [15] we presented a peer-to-peer load balancing architecture, focusing on reactive, node-oriented hot-spot detection that does not utilize prediction. Load shedding [4, 21, 23] has been ex-

plored before as a means to alleviate application hot-spots in stream processing systems. Our goal when alleviating application hot-spots via migration is to do so in a less intrusive manner. Similar to our work, [23] identifies the need for proactive QoS management and proposes operator selectivity estimation using sampling. Their methods however refer to centralized stream processing on a single node.

Workload prediction has been studied in various contexts and [17] discusses how some workloads have been shown to be most accurately represented by open models, while others by closed ones. Dinda [7] has shown the effectiveness of linear models in predicting host load, network bandwidth, and performance data. In the domain of grid computing multi-resource prediction has been proposed [11], where the processor utilization is cross-correlated with the memory utilization. We also utilize cross-correlation, but between different nodes rather than between different resources. Performance prediction for multi-tier web servers [19, 26] is also relevant to our work, provided that all tiers are considered and not just one which is assumed to be the bottleneck. [19] proposes a model based on queuing theory, to predict performance as a function of the transaction mix. For stream processing applications however, rate fluctuations rather than the type of required processing affect performance. For the same reason, certain assumptions regarding the distribution of arrival rates that are needed for queueing analysis, may not hold. [26] proposes a model based on regression to predict the processing cost of web transactions and drive capacity planning decisions. We also employ linear regression but focus on online execution time prediction.

## 7. Conclusions

We have described hot-spot prediction and alleviation mechanisms for distributed stream processing applications. Our algorithms for hot-spot prediction are based on the statistical methods of linear regression and correlation, utilizing only light-weight, passive measurements. Statistics collection and hot-spot prediction and alleviation are carried out at run-time by all nodes independently, building upon a fully decentralized architecture. The experimental evaluation of our techniques on the Synergy middleware over PlanetLab, and using a real network monitoring application operating on traces of real TCP traffic, demonstrated high prediction accuracy and substantial performance benefits with moderate monitoring and migration overheads.

## References


[1] Y. Ahmad and U. Çetintemel. Network-aware query processing for stream-based applications. In *VLDB*, 2004.

[2] R. Arpaci-Dusseau. Run-time adaptation in river. *ACM Transactions on Computer Systems*, 21(1):36–86, Feb. 2003.

[3] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *NSDI*, 2004.

[4] P. Barlet-Ros et al. Load shedding in network monitoring applications. In *USENIX Annual Technical Conference*, 2007.

[5] A. Bavier et al. Operating systems support for planetary-scale network services. In *NSDI*, 2004.

[6] F. Chen, T. Repantis, and V. Kalogeraki. Coordinated media streaming and transcoding in peer-to-peer systems. In *IPDPS*, 2005.

[7] P. Dinda. Design, implementation, and performance of an extensible toolkit for resource prediction in distributed systems. *IEEE TPDS*, 17(2):160–173, February 2006.

[8] X. Gu, P. Yu, and K. Nahrstedt. Optimal component composition for scalable stream processing. In *ICDCS*, 2005.

[9] L. Kleinrock. *Queueing Systems. Volume 1: Theory*. John Wiley and Sons Inc., New York, NY, USA, 1975.

[10] V. Kumar, B. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan. Resource-aware distributed stream management using dynamic overlays. In *ICDCS*, 2005.

[11] J. Liang, K. Nahrstedt, and Y. Zhou. Adaptive multi-resource prediction in distributed resource sharing environment. In *CCGRID*, 2004.

[12] D. Montgomery and G. Runger. *Applied Statistics and Probability for Engineers*. John Wiley & Sons Inc., NY, 2006.

[13] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[14] T. Repantis, X. Gu, and V. Kalogeraki. Synergy: Sharing-aware component composition for distributed stream processing systems. In *Middleware*, 2006.

[15] T. Repantis and V. Kalogeraki. Alleviating hot-spots in peer-to-peer stream processing environments. In *DBISP2P*, 2007.

[16] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[17] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *NSDI*, 2006.

[18] M. Shah, J. Hellerstein, S.Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.

[19] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *EuroSys*, 2007.

[20] Stream Query Repository. http://infolab.stanford.edu/stream/sqr/netmon.html, 2002.

[21] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, 2003.

[22] The Internet Traffic Archive. http://ita.ee.lbl.gov/html/contrib/lbl-tcp-3.html, 1994.

[23] Y. Wei, V. Prasad, S. Son, and J. Stankovic. Prediction-based QoS management for real-time data streams. In *RTSS*, 2006.

[24] Y. Xing, J. Hwang, U. Cetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, 2006.

[25] Y. Xing, S. Zdonik, and J. Hwang. Dynamic load distribution in the Borealis stream processor. In *ICDE*, 2005.

[26] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier appications. In *ICAC*, 2007.