Supporting Self-Adaptation in Streaming Data Mining Applications

Liang Chen Gagan Agrawal
Department of Computer Science and Engineering
Ohio State University, Columbus OH 43210
{chenlia,agrawal}@cse.ohio-state.edu

ABSTRACT

There are many application classes where the users are flexible with respect to the output quality. At the same time, there are other constraints, such as the need for real-time or interactive response, which are more crucial. This paper presents and evaluates a runtime algorithm for supporting adaptive execution for such applications. The particular domain we target is distributed data mining on streaming data. This work has been done in the context of a middleware system called GATES (Grid-based AdapTive Execution on Streams) that we have been developing.

The self-adaptation algorithm we present and evaluate in this paper has the following characteristics. First, it carefully evaluates the long-term load at each processing stage. It considers different possibilities for the load at a processing stage and its next stages, and decides if the value of an adaptation parameter needs to be modified, and if so, in which direction. To find the ideal new value of an adaptation parameter, it performs a binary search on the specified range of the parameter.

To evaluate the self-adaptation algorithm in our middleware, we have implemented two streaming data mining applications. The main observations from our experiments are as follows. First, our algorithm is able to quickly converge to stable values of the adaptation parameter, for different data arrival rates, and independent of the specified initial value. Second, in a dynamic environment, the algorithm is able to adapt the processing rapidly. Finally, in both static and dynamic environments, the algorithm clearly outperforms the algorithm described in our earlier work and an obvious alternative, which is based on linear-updates.

1. INTRODUCTION

In recent years, there has been much interest on adaptive or *autonomic* computing. Adapting applications or programs has been studied by many, and a variety of solutions have been proposed, including those through new algorithms [17], runtime/middleware [14, 5, 3, 27], and language/compilers [12, 9, 11].

There are many application classes where the users are flexible with respect to the output quality. At the same time, there are other constraints, such as the need for real-time response, or limit on the consumption of certain resources, which are more crucial. For example, while visualizing simulation data, the output can be viewed at different granularity, i.e, the output image can be 512×512, 1024×1024, or 2048×2048, etc. While it is preferable to view the image at the finest level, constraints such as the need for real-time response or interactivity could be more important. Examples of applications where the users can have some flexibility in the output arise in multimedia (including video/audio streaming applications), image processing, scientific visualization, and data mining/analysis on simulation data.

This paper presents and evaluates a runtime algorithm for supporting adaptive execution. The particular domain we target is distributed data mining on streaming data. This work has been done in the context of a middleware system called GATES (Grid-based AdapTive Execution on Streams) that we have been developing [7]. GATES system has been designed to support processing of distributed data streams in a wide-area environment.

In the stream model of processing, data arrives continuously and needs to be processed in *real-time*, i.e., the processing rate must match the arrival rate. In view of this, an important goal of the GATES system is to allow the most accurate analysis, while still meeting the real-time constraint. To enable this, application developers can expose one or more *adaptation* parameters, along with a range of their acceptable values. A higher (or lower) value of the adaptation parameter results in more accurate but slower processing. Thus, the goal of the system is determine the highest (or the lowest) value of the parameter which can still meet the real-time constraint. Moreover, the environment for processing the data streams can be *dynamic*, i.e., data arrival rates, and/or the available network bandwidth or CPU cycles can vary over time. In such cases, the system should be able to adjust adaptation parameters dynamically. Such functionality is achieved through a *self-adaptation* algorithm.

The self-adaptation algorithm we present and evaluate in this paper has the following characteristics. First, it carefully evaluates the long-term load at each processing stage. It consider different possibilities for the load at a processing stage and its next stages, and decides if the value of an adaptation parameter needs to be modified, and if so, in which direction. To find the ideal new value of an adaptation parameter, it performs a binary search on the specified range of the parameter.

To evaluate the self-adaptation algorithm in our middleware, we have implemented two streaming data mining applications. These are, clustering evolving data streams [1], and finding frequent items in distributed data streams [18]. GATES' support for specifying and deploying the processing as a series of stages simplifies the development of these streaming data mining applications. Moreover, we also show how each of these applications naturally has an *adaptation* parameter, which allows a trade-off between accuracy and processing rate.

We have evaluated our self-adaptation algorithm extensively using these two applications. The main observations from our experiments are as follows. First, our algorithm is able to quickly converge to stable values of the adaptation parameter, for different data arrival rates, and independent of the initial value that is specified. Second, in a dynamic environment, the algorithm is able to adapt the processing rapidly. Finally, in both static and dynamic environments, the algorithm clearly outperforms the algorithm described in our earlier work [7], and an obvious alternative, which is based on linear-updates.

1-4244-0054-6/06/$20.00

2. OVERVIEW OF THE GATES SYSTEM

This section describes the motivation and the major design aspects of the GATES system.

2.1 Motivation

Increasingly, a number of applications across computer sciences and other science and engineering disciplines rely on, or can potentially benefit from, analysis and monitoring of *data streams*. In the stream model of processing, data arrives continuously and needs to be processed in *real-time*, i.e., the processing rate must match the arrival rate. There are two trends contributing to the emergence of this model. First, scientific simulations and increasing numbers of high precision data collection instruments (e.g. sensors attached to satellites and medical imaging modalities) are generating data continuously, and at a high rate. The second is the rapid improvements in the technologies for Wide Area Networking (WAN), as evidenced, for example, by the National Lambda Rail (NLR) effort and the interconnectivity between the TeraGrid and Extensible Terascale Facility (ETF) sites. As a result, often the data can be transmitted faster than it can be stored or accessed from disks within a cluster.

The important characteristics that apply across a number of stream-based applications are: 1) the data arrives continuously, 24 hours a day and 7 days a week, 2) the volume of data is enormous, typically tens or hundreds of gigabytes a day, and the desired analysis could also require large computations, 3) often, this data arrives at a distributed set of locations, and all data cannot be communicated to a single site, 4) it is often not feasible to store all data for processing at a later time, thereby, requiring analysis in *real-time*.

We briefly describe two representative examples. The first application we consider is online network intrusion detection, which is a critical step for cyber-security. Online analysis of streams of connection request logs and identifying unusual patterns is considered useful for network intrusion detection [10]. To be really effective, it is desirable that this analysis be performed in a distributed fashion, and connection request logs at a number of sites be analyzed. Large volumes of data and the need for real-time response make such analysis challenging. The second example is computer vision based surveillance. Multiple cameras shooting images from different perspectives can capture more information about a scene or a set of scenes. This can enable tracking of people and monitoring of critical infrastructure [4]. A recent report indicated that real-time analysis of the capture of more than three digital cameras is not possible on current desktops, as the typical analysis requires large computations. Distributed and grid-based processing can enable such analysis, especially when the cameras are physically distributed and/or high bandwidth networking is available.

We view the problem of flexible and adaptive processing of distributed data streams as a grid computing problem. We believe that a distributed and networked collection of computing resources can be used for analysis or processing of these data streams. Computing resources close to the source of a data stream can be used for initial processing of the data stream, thereby reducing the volume of data that needs to be communicated. Other computing resources can be used for more expensive and/or centralized processing of data from all sources. Because of the real-time requirements, there is a need for adapting the processing in such a distributed environment, and achieving the best accuracy of the results within the real-time constraint.

2.2 Key Goals

There are three main goals behind the design of the system.

1. Enable the application to achieve the best accuracy, while maintaining the *real-time* constraint. For this, the middleware allows the application developers to expose one or more *adaptation* parameters at each stage. An *adaptation* parameter is a tunable parameter whose value can be modified to increase the processing rate, and in most cases, reduce the accuracy of the processing. Examples of such adaptation parameters are, rate of sampling, i.e., what fraction of data-items are actually processed, and size of summary structure at an intermediate stage, which means how much information is retained after a processing stage. The middleware automatically adjusts the values of these parameters to meet the real-time constraint on processing. This is achieved through a *self-adaptation* algorithm, which is also the focus of this paper.

2. Support distributed processing of one or more data streams, by facilitating applications that comprise a set of *stages*. For analyzing more than one data stream, at least two stages are required. Each stage accepts data from one or more input streams and outputs zero or more streams. The first stage is applied near sources of individual streams, and the second stage is used for computing the final results. However, based upon the number and types of streams and the available resources, more than two steps could also be required. All intermediate stages take one or more intermediate streams as input and produce one or more output streams. GATES's APIs are designed to facilitate specification of such stages.

3. Enable easy *deployment* of the application. This is done by supporting a *Launcher* and a *Deployer*. The system is responsible for initiating the different stages of the computation at different resources. The system also allows the use of existing grid infrastructure. Particularly, the current implementation is built on top of the Open Grid Services Infrastructure (OGSI) [13], and uses its reference implementation, Globus Toolkit (GT) 3.0.

3. SELF-ADAPTATION ALGORITHM

This section describes the self-adaptation algorithm we have implemented and evaluated in the GATES system. As we stated in the previous section, the goal of this algorithm is to modify the value(s) of adaptation parameter(s) at runtime, so as to achieve highest level of accuracy while still meeting the real-time constraint. While the basic framework and some of the metrics used are identical to the adaptation algorithm presented in our earlier work [7], the algorithm presented here is distinct in the following ways. First, it systematically considers different possibilities at a particular stage and its successor. Second, this algorithm does not require that certain functions and parameters be tuned for a particular application. Finally, it is able to converge to a stable value of the adaptation parameter faster.

3.1 Background

An application built on the GATES middleware comprises a set of pipelined stages. By modeling every stage as a server and viewing the input buffer of a stage as a queue of the server, we can get a queuing network model of the system. As an example, the model of an application with three stages is shown in Figure 2.

GATES applications are required to use a specific API to expose

Symbols	Definition
Variables	
d	Current length of the queue
$\bar{d}$	Average of the d values in recent times
$\tilde{d}$	Long-term average queue size factor
t_1	The number of times system was *over-loaded*
t_2	The number of times system was *under-loaded*
w	The difference in the number of times system was recently *under/over-loaded*
ϕ_1	Functions reflecting queue's long-term load
ϕ_2, ϕ_3	Functions reflecting queue's recent load
$\mathcal{P}$	Adaptation parameter for a server
Constants	
α	Learning rate for $\tilde{d}$
W	Window size
E	Expected length of the queue
L	Capacity of the queue
P_1, P_2, P_3	Weights to ϕ_1, ϕ_2, ϕ_3, respectively
LT	Maximum (Minimum) threshold for the load

Figure 1: Summary of Symbols Used

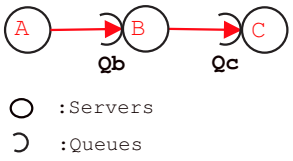

Figure 2: A Queuing model of an Application with Three Stages

adaptation parameters. Specifically, the function

specifyPara(init_value, max_value,min_value, incre_or_decre)

is used to specify an initial value and a range of acceptable values of an adaptation parameter, and also state whether increasing the parameter value results in faster or slower processing.

Assume that the data arrives at a server in fixed-size packets. Let the average data arrival rate be denoted by λ. The rate at which the server is able to consume the packets is denoted by μ. If we have flexibility in controlling the accuracy of the analysis, our goal is to adjust the parameters to maintain a good balance between λ and μ. Clearly, if $\mu < \lambda$, the queue will saturate, and real-time constraint on processing cannot be met. In this case, we need to slow-down the processing that is performed by the sending server, i.e., make the processing more accurate. Alternatively, we can increase the rate of processing at the current server, possibly losing some accuracy. If λ is much lower than μ, we are under-utilizing the current server. In this case, we can speed up the processing at the sending server.

As λ and μ cannot be determined at any given instance, we focus on the current length of the queue, which is indicative of the ratio between the two. Our objective is to keep the average queue size within an *interval* between the two pre-defined thresholds. This goal could be achieved by dynamically adjusting the processing rates of the current and the preceding server. This, in turn, can be done by properly tuning the value of adaptation parameters.

Overall, there are two challenges in the algorithm. The first challenge is to correctly weigh in the recent as well as long-term behavior of the queue. For this purpose, we introduce a *long-term average queue size* factor, denoted by $\tilde{d}$. The other challenge is to promptly have the adaptation parameter converge to an ideal value. This will allow the algorithm to be sensitive to a varying environment. The main two steps of the algorithm, evaluating $\tilde{d}$, and adjusting parameters, respectively, overcome these two challenges. The list of terms used in our algorithm is listed in Figure 1.

Evaluating Long-Term Load: This calculation is based upon three distinct *load factors* and learning by weighing these factors. These three load factors are denoted by ϕ_1, and ϕ_2 and ϕ_3, respectively.

ϕ_1 focuses on long-term load and is computed as follows. If the current length of the queue, d, is larger or less than some thresholds, we say that the queue is *over* or *under*-loaded. From the start of the system, t_1 is the number of times the system was found to be *over-loaded* and t_2 is the number of times the system was found to be *under-loaded*. t_1 and t_2 describe the long-term behavior of the system. We compute ϕ_1 as follows.

$$\phi_1(t_1, t_2) = \begin{cases} \frac{t_1 - t_2}{t_1 + t_2} & \text{if } (t_1 + t_2 > 0) \\ 0 & \text{if } (t_1 + t_2 = 0) \end{cases}$$

The reason for choosing this expression is that even if t_1 and t_2 are large, if they are very close, we believe that the system is properly loaded from a long-term perspective.

To focus on the short-term behavior, we define the variable w and $\bar{d}$. We choose a window size W and record the last W times the system was observed to be over or under-loaded. w is a variable that is incremented by 1 for every occurrence of over-load within the window, and decremented by 1 for every occurrence of under-load within this window. Thus, $|w| \leq W$. We compute ϕ_2 as follows.

$$\phi_2(w) = \begin{cases} w \times \frac{1}{|w|} \times e^{\frac{|w| - W}{2}} & \text{if } (|w| \neq 0) \\ 0 & \text{if } (|w| = 0) \end{cases}$$

We also use another short-term indicator, which is computed as follows. $\bar{d}$ is the average of the d values observed in recent times. Furthermore, E is a user-defined expected length of the queue and L is the capacity of the queue. Then, we have

$$\phi_3(\bar{d}) = \begin{cases} \frac{\bar{d} - E}{E} & \text{if } \bar{d} < E \\ \frac{\bar{d} - E}{L - E} & \text{if } \bar{d} \geq E \end{cases}$$

The range of values of $\phi_i (i = 1, 2, 3)$ is $[-1, 1]$. Moreover, the closer $|\phi_i|$ is to 1, it is more likely that the unit is over or under-loaded. Now, we can use the following equation to calculate $\tilde{d}$.

$$\tilde{d} = \alpha \times \tilde{d} + (1 - \alpha) \times (P_1 \times \phi_1(t_1, t_2) + P_2 \times \phi_2(w) + P_3 \times \phi_3(\bar{d}))$$

Here, P_1, P_2, P_3 are the factors that give weight to ϕ_1, ϕ_2, and ϕ_3, respectively, and satisfy the constraint $P_1 + P_2 + P_3 = 1$. Moreover, $0 < \alpha < 1$ is a pre-defined learning rate which helps remove transient behavior.

Similar to ϕ_i, $\tilde{d} \in [-1, 1]$, and the closer $|\tilde{d}|$ to 1, it is more likely that the unit is having very high or low load. Particularly, when $\tilde{d}$ exceeds the pre-defined interval $[-LT, LT]$, the current server will be thought of as being under-loaded or over-loaded. These load states indicated by $\tilde{d}$ are used to tune adaptation parameters.

3.2 New Algorithm and Parameter Adjustment

State	Condition		Adjustment Strategy
1	$\tilde{d_B} < -LT$,	$\tilde{d_C} < -LT$	Decrease $\mathcal{P}_B$ and increase accuracy
2	$\tilde{d_B} < -LT$,	$-LT \leq \tilde{d_C} \leq LT$	Decrease $\mathcal{P}_B$ and increase accuracy
3	$\tilde{d_B} < -LT$,	$\tilde{d_C} > LT$	Decrease $\mathcal{P}_B$ and increase accuracy
4	$-LT \leq \tilde{d_B} \leq LT$,	$\tilde{d_C} < -LT$	Do not change $\mathcal{P}_B$
5	$-LT \leq \tilde{d_B} \leq LT$,	$-LT \leq \tilde{d_C} \leq LT$	Do not change $\mathcal{P}_B$
6	$-LT \leq \tilde{d_B} \leq LT$,	$\tilde{d_C} > LT$	Decrease $\mathcal{P}_B$ and increase accuracy
7	$\tilde{d_B} > LT$,	$\tilde{d_C} < -LT$	Increase $\mathcal{P}_B$ to speed up the processing rate
8	$\tilde{d_B} > LT$,	$-LT \leq \tilde{d_C} \leq LT$	Do not change $\mathcal{P}_B$
9	$\tilde{d_B} > LT$,	$\tilde{d_C} > LT$	Decrease $\mathcal{P}_B$ to and increase accuracy

Figure 3: Summary of *Load States*

We now describe our new algorithm, focusing on how an adaptation parameter is modified at runtime according to the evaluation of $\tilde{d}$. In the following discussion, we specifically focus on the server B in Figure 2. The server B receives data from the server A and processes and forwards data to the server C. There could be two types of adaptation parameter $\mathcal{P}_B$ at the server B. One is called *performance parameter*. Incrementing its value results in increasing the processing rate, μ_B, and decreasing the accuracy of the processing. Thus, a higher value of $\mathcal{P}_B$ will allow the server B to process the data faster. However, this will also result in a higher load for the server C. Another type of adaptation paramenter is *accuracy parameter*. Contrary to a performance parameter, changing the value of an accuracy parameter can contribute to the reverse impacts as we describe above. For simplicity, our description of the algorithm assumes that the server B has a performance parameter. We have considered both accuracy and performance parameters in our experimental evaluation of the algorithm.

There are two main issues for the parameter adaptation component of our algorithm. The first question is deciding when we should increase or decrease $\mathcal{P}_B$, and when we should leave it unchanged. The second question is deciding the new value of $\mathcal{P}_B$, when we need to change the parameter. To answer the first question, we define the *load state* for a given server.

DEFINITION 1. *The* load state *for the server B, denoted by S_B, is based on the tuple $(\tilde{d_B}, \tilde{d_C})$. Particularly, we consider three possibilities for each of $\tilde{d_B}$ and $\tilde{d_C}$, which are, $\tilde{d} < -LT$, $-LT \leq \tilde{d} \leq LT$ or $\tilde{d} > LT$. Thus, there are nine distinct load states for a server.*

The nine possible load states are shown in Figure 3. Among these states, we consider S_4, S_5 and S_8 *convergent* states. Each of the other 6 states is considered *non-convergent*. In a convergent state, there is no need to modify the adaptation parameter at the server B, whereas in a non-convergent state, the parameter $\mathcal{P}_B$ needs to be modified.

The action taken by our algorithm in each of these cases is shown in Figure 3. We now explain the rationale for the strategy for different cases.

Initially, we consider states S_1, S_2, and S_3. These three states are common in the fact that the server B is underutilized. In such a case, we can improve the accuracy of processing at the server B, irrespective of the state of the server C. Note that in a streaming environment, the data arrival rate at the first stage cannot be modified. Thus, for each of the subsequent stages, our goal is to be able to achieve highest possible accuracy, without making them the bottleneck.

Next, let us consider the convergent states, which are S_4, S_5 and S_8. When the server B's state is S_4, it implies that the server B's load is within the desired range, whereas, the server C is under-utilized. In such a case, the server B does not need to make any change. Note, however, that the same algorithm is applied on the server C also, and in this case, server C can increase the accuracy of the processing, if it has a parameter to modify. In the state S_5, the load at both B and C is in the desired range, so clearly, no changes are required. The last convergent state is S_8. In this state, the server B is overloaded, but the server C is not underloaded. Thus, increasing the processing rate at the server B can make C overloaded, which is not desirable. Thus, the server B does not make a change. Again, note that the same algorithm is being applied at the server A, and if possible, server A should decrease the rate at which it forwards data to the server B.

The three remaining states are S_6, S_7 and S_9. In the state S_6, the server C is overloaded. In this case, the server B will adjust to slow down the rate at which it forwards the data to the server C. Note that it may be possible for C to avoid the overload by adjusting a parameter locally, but the server B does not assume that such a parameter exists. The action for the state S_7 is easy to explain. The server B is overloaded, whereas the server C is underloaded. Thus, B needs to increase the rate of processing. The state S_9 is quite challenging, as both B and C are overloaded. In this case, the only likely acceptable solution will be to have the server A slow down the processing. To facilitate this, the server B further slows down the processing. This will reduce the load at C, but can increase the load at B. Since the server A views the load information at the server B, and not the server C, this is most likely to create convergence.

The second important challenge for our algorithm is to determine the new (higher or lower) value for $\mathcal{P}_B$, when a change in its value is needed. There are several considerations that must be used. First, the system should be able to converge to an *ideal value* for the adaptation parameters, i.e., the one which allows the best accuracy, while still meeting the real-time constraint. Second, this convergence should be achieved quickly. This is important for adapting in a dynamic environment, and for avoiding loss of packets when buffer sizes are small.

One obvious way for adjusting parameters will be to use a *linear update* algorithm, i.e., changing $\mathcal{P}_B$ by a fixed value in each iteration. As we will show through our experimental evaluation, this scheme has the following shortcoming. If the amount of the change is large, the system may never reach the ideal value. On the other hand, if the amount of the change is small, a large number of iterations may be required for convergence.

Therefore, we have designed a method which is similar to a binary tree search. The overall algorithm is shown in Figure 4. Two variables, $left\_border$ and $right\_border$, are used in the third step of the algorithm. These define a range within which $\mathcal{P}_B$ can be changed.

Algorithm getSuggestedPara()

1. //Calculate the tuple T
   $\tilde{d_B}$ = calcuateMyLongTermLoad();
   next_server = getMyFollowingServer();
   $\tilde{d_C}$ = next_server.calcuateMyLongTermLoad();
   $T = (\tilde{d_B}, \tilde{d_C})$
2. Determine which load state T belongs to
3. //Update the adaptation parameter's value
   if the direction is "do not change $\mathcal{P}_B$"
   {
   // the current value of $\mathcal{P}_B$ is val
   do nothing
   }
   if the direction is "decrease $\mathcal{P}_B$"
   {
   //The initial value of $right\_border$ is max_value
   $right\_border = val$
   $val = \frac{left\_border + right\_border}{2}$
   }
   if the direction is 'increase $\mathcal{P}_B$'
   {
   //The initial value of $left\_border$ is min_value
   $left\_border = val$
   $val = \frac{left\_border + right\_border}{2}$
   }
4. return val

Figure 4: Self Adaptation Algorithm

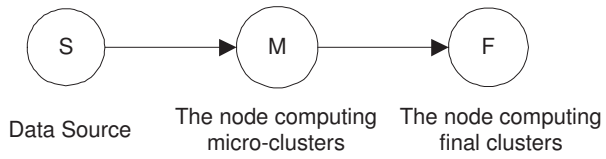

Figure 5: Communication Topology for the CluStream Application

Their initial values are the minimum and the maximum values of the adaptation parameter, which the GATES API requires from application developers. These values are denoted as min_value and max_value, respectively.

We proceed as follows. The current value is denoted as val. When it is determined that $\mathcal{P}_B$ should be decremented, the new range is changed to $[left\_border, val]$, and val is updated to the mid-point of the new range. When it is determined that $\mathcal{P}_B$ should be incremented, the new range is $[val, right\_border]$, and again, val is updated to the mid-point of the new range.

Therefore, $\mathcal{P}_B$ will eventually converge to a value within the range $[max\_value - min\_value]$, assuming that the environment is static. In practice, our algorithm only requires between 3 and 5 steps to converge in a static environment.

Finally, we consider a *dynamic* environment. An environment is *dynamic* if the data arrival rate, available network bandwidth, and/or CPU cycle availability is varying. In such a case, the algorithm needs to be able to determine new ideal value. We modify the algorithm as follows. We store all previous ranges in a stack. When $\mathcal{P}_B$ needs to be changed, and if current range is very small compared with the initial range, the previous range is popped from the stack.

4. STREAMING DATA MINING APPLICATIONS

This section describes the two streaming data mining applications that were used for our experimental study. We show how these applications can be developed and deployed using GATES system's support. We also show how each of these applications naturally has an adaptation parameter, which allows trade-off between processing rate and accuracy.

The first application is clustering evolving data streams [1], and is referred to as *CluStream*. Clustering involves grouping similar object or data points from a given set into *clusters*. The particular problem considered here is clustering data arriving in continuous streams, especially as the distribution of data can change over time.

The algorithm we consider [1] approaches the problem as follows. The clustering process is divided into two major steps. The first steps involves computing *micro-clusters* that summarize statistical information in a data stream. The second step uses micro-clusters to compute the final clusters.

This two-step clustering algorithm can be easily implemented using the GATES middleware. Figure 5 shows the three stages that are used. The first stage is simply the data source, which sends streaming data to the second stage. The second stage computes micro-clusters. After a certain number of data points have been processed, it sends the computed micro-clusters to the third stage. The third and the final stage then applies the modified k-means algorithm [1] to create and output the final clusters. To make the system more efficient, the GATES transmits data streams through TCP sockets and only uses SOAP messages for stages to exchange load states.

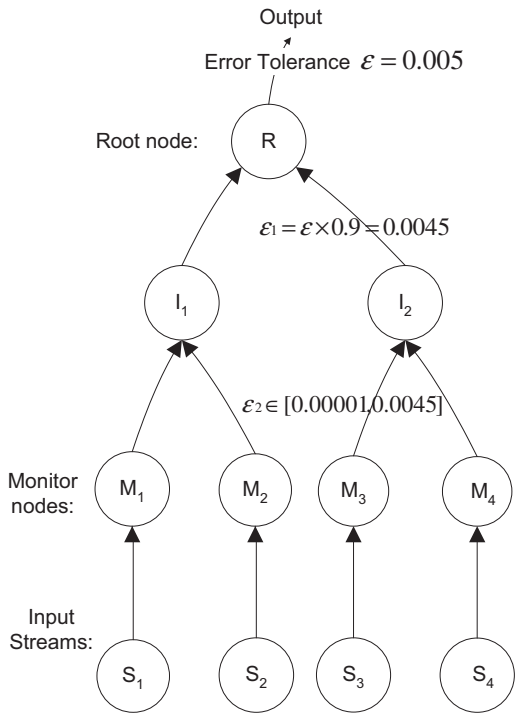

Figure 6: Communication Topology for the Dist-Freq-Counting Application

Note that the final number of clusters desired is specified by the user. However, the number of micro-clusters computed by the second stage needs to be chosen by the algorithm. A larger number of micro-clusters results in better accuracy in computing the final clusters. But, the amount of computation at the second stage and the volume of communication between the second and third stages are both directly proportional to the number of micro-clusters. Thus, the number of micro-clusters becomes an *accuracy parameter* for this application.

The second application we have studied finds frequent occurring itemsets in a distributed data stream and is referred to as *Dist-Freq-Counting* [18]. The problem is of finding frequently occurring itemsets across a set of data streams. If the distribution of data across the different streams is different, and if the communication bandwidth is limited, this problem can be quite challenging.

The algorithm we consider is extention of a proposed algorithm for finding frequent items from distributed streams [18]. The algorithm addresses the problem stated above by arranging the nodes in a hierarchical structure. Figure 6 shows an example of such a structure. Each *monitor* node M_i counts the frequencies of itemsets appearing in the stream S_i, and periodically sends this information to its parent node, which could be an intermediate node or the root node. Intermediate nodes combine the frequency information received from their children and pass them up to their parent node. Finally, the root node outputs the itemsets whose frequencies exceed the specified support threshold τ.

To reduce communication loads, the monitor and intermediate nodes should avoid sending less frequent itemsets over the links. Therefore, the algorithm uses an error tolerance parameter ϵ at every node, except the data sources. Only the itemsets with frequency greater than this parameter are forwarded to the next node.

The value of a tolerance parameter impacts both the processing rate and the accuracy. With a higher value, we could miss itemsets which may be frequent overall. With a lower value, the volume of communication can be increased. Generally, it is desirable that all nodes at the same level use the same tolerance value, and the tolerance value (frequency) is increased as we move closer to the root node.

In our implementation, we consider the tolerance parameter at the monitor nodes as a *performance parameter*. This is because the communication volume at this stage can be the highest, and therefore, the tolerance parameter has the largest impact on the performance.

5. EXPERIMENTAL EVALUATION

This section presents results from a number of experiments we conducted to evaluate our self-algorithm and the use of the GATES middleware for streaming data mining applications. Specifically, we had the following goals in our experiments:

- Demonstrate that our self-adaptation algorithm is able to quickly converge to the ideal value of the adaptation parameter, for different data stream arrival rates.
- Show that the algorithm is not sensitive to the initial value of adaptation parameter.
- Show that how the algorithm is able to vary the value of an adaptation parameter as the execution environment changes dynamically.
- Show that our algorithm is more efficient and effective than the algorithm presented in our earlier work [7] and an obvious alternative, which involves the use of linear adjustments.

The experiments were conducted using the two streaming data mining applications described in the previous section. For the CluStream application, we used the KDD-CUP'99 Network Intrusion Detection dataset. For Dist-Freq-Counting, we use a dataset generated by the IBM data generator [2]. The average size of each transaction in this dataset is 6.

5.1 Experimental Environment

For efficient and distributed processing of streaming data in a grid environment, we need high bandwidth networks and a certain level of quality of service support. Recent trends are clearly pointing in this direction. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single linux cluster. Each node in the cluster has a Pentium III 933MHz CPU with 512MB of main memory and 300GB local disk space, interconnected with switched 100 Mb/s Ethernet. We sampled the queue loads every 50 ms and recalculate the values of the variables in Figure 1. Application developers can specify those pre-defined constants in Figure 1 to meet applications' specific needs. For instance, if an application needs a bigger buffer and the system can afford it, the developer can set L to a larger value. The developer could set P_1 larger than $P_2 + P_3$ if they believe the application should focus on the system's long term behavior. In our experiments, we configured the parameter values as indicated in the Figure 7.

We conducted 3 sets of experiments, which are described in the rest of this section.

5.2 Convergence In a Static Environment

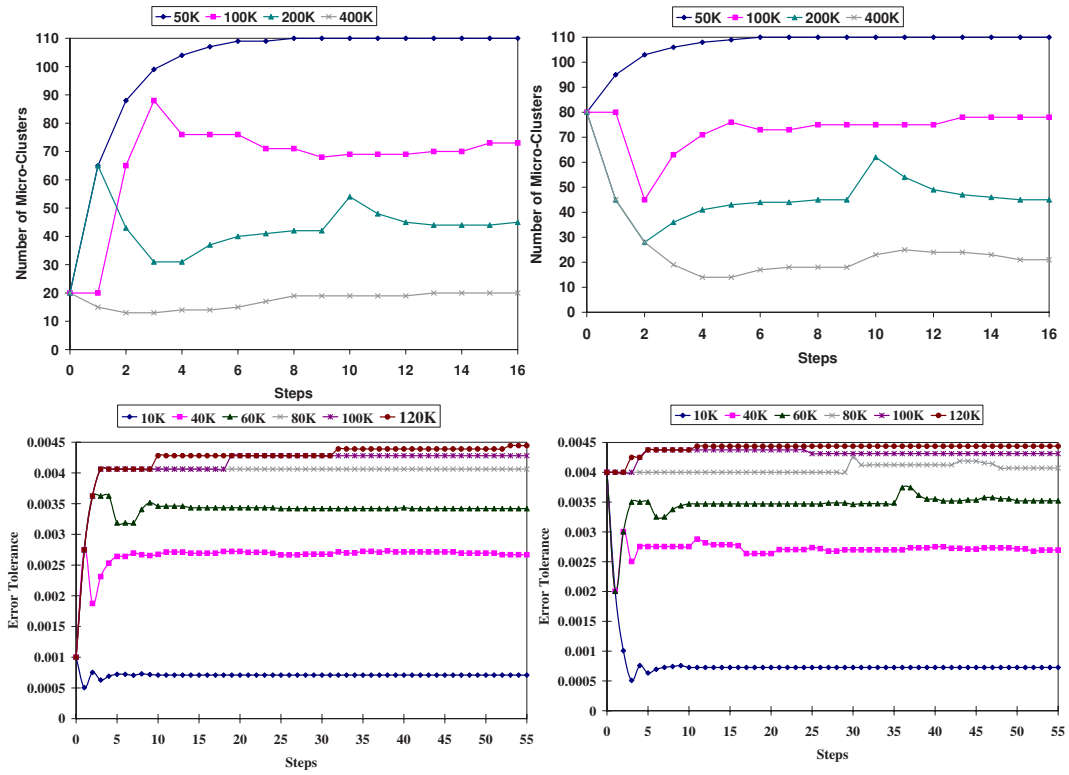

Figure 8: Convergence Under Different Data Arrival Rates: CluStream Application (top two charts) and Dist-Freq-Counting (bottom two charts)

Parameter	Value
α	0.9
W	5
E	0.65
L	20,000
P_1, P_2, P_3	0.3, 0.5, 0.2, respectively
LT	0.85

Figure 7: Parameter Values

Our first experiment demonstrated that the self-adaptation algorithm can choose the ideal values for the adaptation parameters under different data arrival rates, irrespective of the initial values of these parameters.

For CluStream, we initialized the number of micro-centers to 20, 40, 80, and 110. The allowed range of this parameter was $[10, 110]$. We used one data source, and controlled the data arrival rate at the second stage to be 50, 100, 200, and 400 Kbps, respectively. The results from the cases where the number of micro-clusters was 20 and 80 are shown as top two charts from Figure 8. Let us consider the first chart, where the initial values is 20. The value of the number of micro-clusters converged to 110, 73, 44, and 20, for the four data arrival rates we considered. The convergence occured in an average of 5 steps, which corresponds to an average of 53 seconds. The X-axis in this chart is the number of *steps*, which denotes the number of invocations of the Algorithm shown in Figure 4. The results are similar when the initial value is 80. A similar set of experiments were also conducted using our second application, Dist-Freq-Counting. The results are shown as the bottom two charts in Figure 8. We set the range of ϵ_2 to be $[0.0001, 0.0045]$. We set all monitor nodes to have the same data arrival rate and then considered six different rates and two different initial values. The algorithm converges in each of the cases.

5.3 Adaptation in a Dynamic Environment

In this subsection, we show that our binary search based adaptation algorithm can quickly adjust the value of an adaptation parameter in a dynamic environment. Such dynamic adaptation may be needed if the data arrival rate varies frequently, and/or if the available network bandwith or CPU cycles can vary. For our experiments, we only considered variations in data arrival rates.

The two charts in Figure 9 consider the CluStream application. The allowed range of number of micro-clusters is $[10, 100]$. The initial value is 60. The initial data arrival rate is 400 Kbps. The data arrival rate is varied with two different frequencies, which are every 120 and 30 seconds, respectively. The left and the right charts in Figure 9 correspond to these two frequencies. The data arrival rate is varied between 40 Kbps and 400 Kbps, with a step of 60 Kbps, applied every 120 or 30 seconds.

The Y-axis in the charts in Figure 9 corresponds to both the data arrival rates, and the number of micro-clusters chosen by our algorithm. The scales for these values are shown in left and right side, respectively, of each chart. Our results show that our algorithm is able to vary the number of micro-clusters with the same frequency as rate of change of data arrival rate. As the data arrival rate increases, the number of micro-clusters goes down to the minimum possible value of 10. As the data arrival rate decreases, it goes back up to a higher value.

One interesting question is, how does the frequency of change of

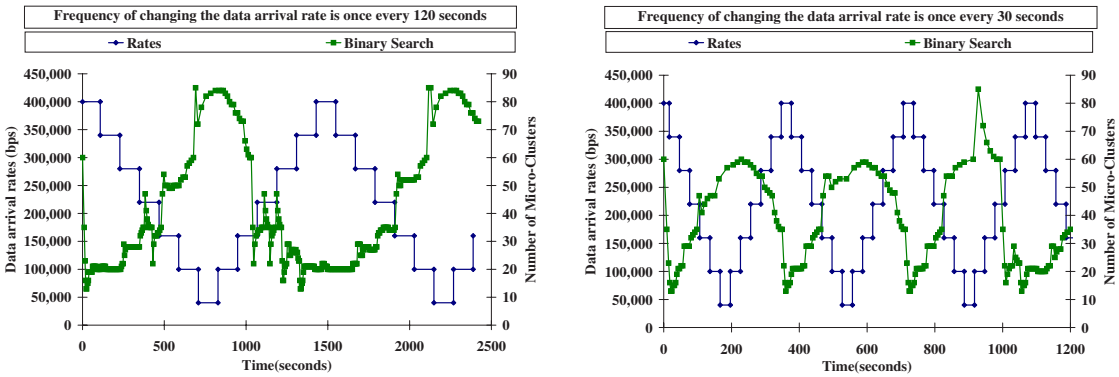

Figure 9: Algorithm Behavior in a Dynamic Environment: CluStream Application

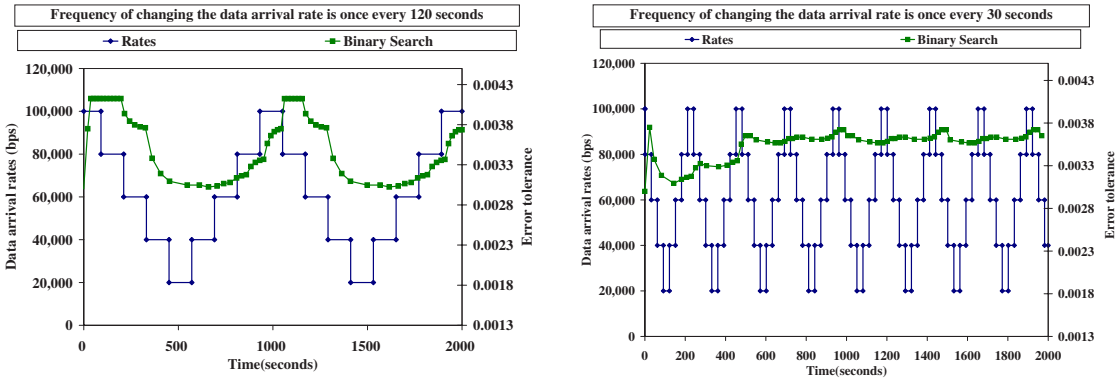

Figure 10: Algorithm Behavior in a Dynamic Environment: Dist-Freq-Counting

data arrival rates impact the algorithm. We can see that the range of number of microclusters is $[10, 85]$ with the slower rate of change (first chart) and $[10, 60]$ with the higher rate of change (second chart). This is because our algorithm needs to see an under-loaded server for a long duration to increase the accuracy of the processing to the highest levels.

We also conducted the same experiment with the other application. The results are shown in Figure 10. The results are very similar. The range of variation of ϵ_2 is quite limited when the frequency in the change of data arrival rates is higher.

5.4 Comparing Different Algorithms

We now compare the self-adaptation algorithm with the obvious alternative, which is a linear update algorithm, and the algorithm presented in our previous work [7]. We compared these algorithms in both static and dynamic environments.

We first consider CluStream. We initially compared the binary search algorithm with the other two algorithms in a static environment. The results are shown in the first chart of Figure 11. For this experiment, the data arrival rate is fixed at 200 Kbps and the number of micro-centers is allowed to vary within the range $[10, 110]$. We consider two scenarios with the binary search algorithm, which corresspond to the initial number of micro-clustering being 60 and 100. We also consider two scenarios with the linear-update algorithm. Though the initial number of micro-clusters is 60 in both the cases, the amount of update in each case is $(110-10)/10 = 10$ and $(110-10)/100 = 1$, respectively.

The binary search algorithm is able to converge to the ideal value of 46 within 6 to 8 steps in the both the cases. With the linear update algorithm with a step of 10, the algorithm never converges, instead, after a few iterations, it starts alternating between 40 and 50. When we use a step of 1, the algorithm converges, but takes nearly 13 steps, or about twice as long as the binary search algorithm. This algorithm shows the main limitation of the linear-update algorithm, which is the difficulty of choosing an appropriate step value. A large value can result in an unstable behaviour, whereas, a small step value can create large delays in convergence. Finally, as we can see from this figure, the number of micro-clusters continuously decreases with the algorithm presented in our earlier work [7]. This is because this algorithm did not consider the possibility of a compute-intensive stage being the bottleneck. We also compared these three algorithms in a dynamic environment. We considered two cases, with the frequency of the change of data arrivals rates being once every 60 seconds and 180 seconds. The results for these two cases are shown in the 2nd and 3rd charts of Figure 11. Again, we considered linear-update algorithm with steps of 10 and 1. The conclusions from these experiments are similar to those from the static experiments. The linear-update algorithm with the step of 10 is unstable and with a step of 1, it is slower to adjust. Our previous algorithm slowly converges to the minimum value of the adaptation parameter. Figure 12 shows the results from the same set of experiments, but using Dist-Freq-Counting. The results are identical.

6. RELATED WORK

We now compare our work with other efforts on support for streaming model of execution, and adaptation through a middleware.

Stream Data Processing: In the area of stream processing, the work that is probably the closest to our work is the dQUOB project [21, 22]. This system enables continuous processing of SQL queries on data streams. Our work is distinct in the following ways. First, we support an API to allow general processing, and not just SQL queries. Second, the processing can be done in a pipeline of stages. Third, it does not support self-adaptation. Stampede is a cluster middleware for supporting streaming applications [24, 25]. Our work is again distinct in consider grid resources and adaptation for real-time processing. Mazzucco *et al.* have looked at the specific support for merging multiple high speed data streams [19].

Adaptation Through a Middleware: Application adaptation has been studied in many contexts, including through (grid) middleware. We briefly survey this work here and state how our work is distinct. As a quick summary, our work is different in focusing on adapting to meet real-time constraint on stream data processing, and in adapting the output of the application to do so.

DART [26] is a system facilitating quick development of adaptive applications. A runtime component is responsible for making adaptation decisions following a set of selected policies. Moura *et al.* present software support in the component-base programming context for construction of auto-adaptive applications [8]. It leaves applications the option to dynamically choose the most beneficial components. ROAM implements resource-aware runtime adaptation for device heterogeneity in mobile systems [14]. Schwan and his group take into account the runtime resource management issues when supporting adaptable applications [23]. Somewhat similar ideas have been considered by Karamcheti and co-workers [6, 16]. Particularly, the notion of *tunable* parameters has similarity to our work, though their focus is not on streaming data. Isert and Schwan have developed a system called ACDS, which includes a monitoring and steering tool for adapting stream based computations [15]. These systems requires that either the resource usage associated with each option be stated explicity or the logic for making adaptation decisions be specified by the application developer. In comparison, we consider a more restrictive class of applications, but automate the adaptation process more.

7. CONCLUSIONS

With increasing focus on interactive and real-time applications in a wide-area environment, it is important for a grid middleware to support adaptive execution, especially, as the execution environment changes dynamically. In this paper, we have considered adaptive execution for stream data processing. Here, the goal of the middleware is to enable the highest level of accuracy, while still maintaining real-time constraint on the processing.

We have developed a self-adaptation algorithm, and have implemented and evaluated it as part of the GATES middleware system. Our algorithm has the following characteristics. First, it carefully evaluates the long-term load at each processing stage. It consider different possibilities for the load at a processing stage and its next stages, and decides if the value of an adaptation parameter needs to be modified, and if so, in which direction. To find the ideal new value of an adaptation parameter, it performs a binary search on the specified range of the parameter.

We have implemented two streaming data mining applications using our middleware, and have extensively evaluated the adaptive capabilities of our middleware. The main observations from our experiments are as follows. First, our algorithm is able to quickly converge to stable values of the adaptation parameter, for different data arrival rates, and independent of the specified initial value. Second, in a dynamic environment, the algorithm is able to adapt the processing rapidly. Finally, in both static and dynamic environments, the algorithm clearly outperforms the algorithm described in our earlier work and an obvious alternative, which is based on linear-updates.

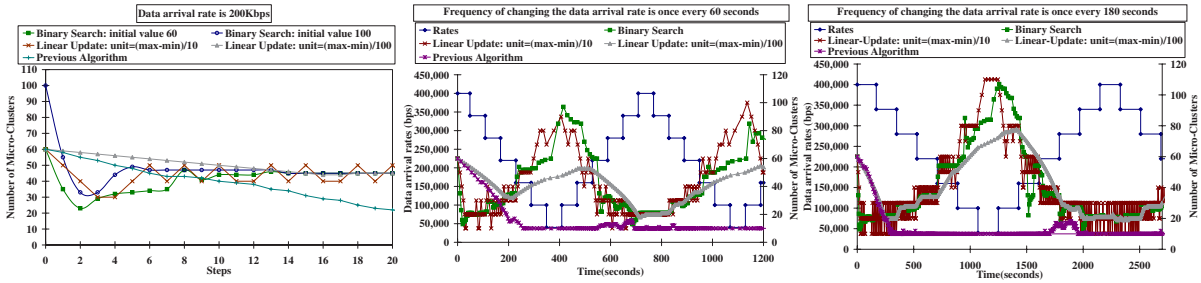

Figure 11: Comparing Different Self-Adaptation Algorithms: CluStream Application

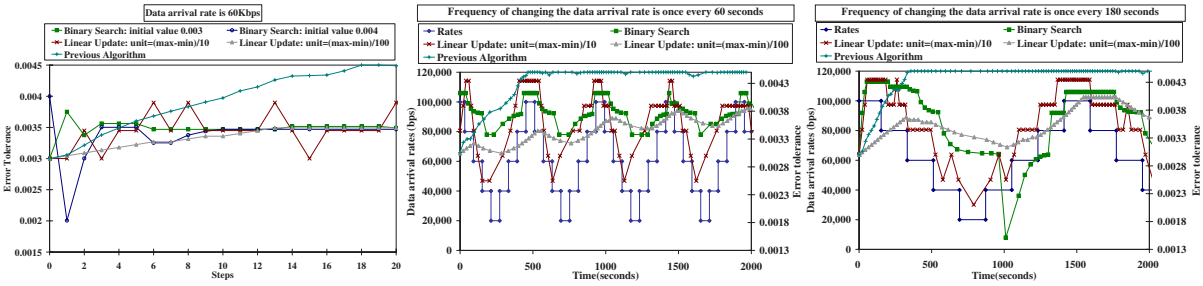

Figure 12: Comparing Different Self-Adaptation Algorithms: Dist-Freq-Counting

8. REFERENCES

[1] C. Aggarwal, J. Han, J. Wang, and P. Yu. A framework for clustering evolving data streams. In *The 29th International Conference on Very Large Data Bases*, 2003.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, 12–15 1994.

[3] Gabrielle Allen, Thomas Dramlitsch, Ian Foster, Tom Goodale, Nick Karonis, Matei Ripeanu, Ed Seidel, and Brian Toonen. Cactus-G Toolkit: Supporting Efficient Execution in Heterogeneous Distributed Computing Environments. Technical report, U. of Chicago, 2001.

[4] E. Borovikov, A. Sussman, and L. Davis. A High-Performance Multi-Perspective Vision Studio. In *Proceedings of the International Supercomputing Conference (ICS)*. ACM Press, June 2003.

[5] Henri Casanova, Graziano Obertelli, Francine Berman, and Rich Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proceedings of Super Computing 2000*, 2000.

[6] F. Chang and V. Karamcheti. Automatic Configuration and Run-time Adaptation of Distributed Applications. In *Proceedings of Conference on High Performance Distributed Computing (HPDC)*, 2000.

[7] Liang Chen, Kolagatla Reddy, and Gagan Agrawal. GATES: A Grid-Based Middleware for Distributed Processing of Data Streams. In *Proceedings of IEEE Conference on High Performance Distributed Computing (HPDC)*. IEEE Computer Society Press, 2004.

[8] Ana Lcia de Moura, Cristina Ururahy, Renato Cerqueira, and Noemi Rodriguez. Dynamic support for distributed auto-adaptive applications. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02)*, July 02 - 05,2002.

[9] Pedro Diniz and Bing Liu. Selector: A Language Construct for Developing Dynamic Applications. In *In the proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, July 2002.

[10] P. Dokas, L. Ertoz, V. Kumar, A. Lazarevic, J. Srivastava, and P. Tan. Data Mining for Network Intrusion Detection . In *Proc. of the NSF Workshop on Next Generation Data Mining*, November 2002.

[11] Wei Du and Gagan Agrawal. Language and Compiler Support for Adaptive Applications. In *Proceedings of Supercomputing 2004*, November 2004.

[12] Brian Ensink, Joel Stanley, and Vikram Adve. Program control language: a programming language for adaptive distributed applications. *J. Parallel Distributed Computing*, 63(11), 2003.

[13] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Open Grid Service Infrastructure Working Group, Global Grid Forum*, June 2002.

[14] Hao hua Chu, Henry Song, Candy Wong, Shoji Kurakake, and Masaji Katagiri. Roam: a seamless application framework. *J. System Software*, 69(3), 2004.

[15] Carsten Isert and Karsten Schwan. ACDS: Adapting computational data streams for high performance. In *14th International Parallel & Distributed Processing Symposium (IPDPS 2000)*, pages 641–646, Cancun, Mexico, May 2000. IEEE Computer Society Press.

[16] Inca-Andreea Ivan, Josh Harman, Michael Allen, and Vijay Karamcheti. Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogeneous Environments. In *Proceedings of the Conference on High Performance Distributed Computing (HPDC)*, June 2002.

[17] Xinyue Li and Han-Wei Shen. Time-critical multiresolution volume rendering using 3d texture mapping hardware. In *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, 2002.

[18] Amit Manjhi, Vladislav Shkapenyuk, Kedar Dhamdhere, and Christopher Olston. Finding (recently) frequent items in distributed data streams. In *The 21st International Conference on Data Engineering*, 2005.

[19] Marco Mazzucco, Asvin Ananthanarayan, Robert L. Grossman, Jorge Levera, and Gokulnath Bhagavantha Rao. Merging multiple data streams on common keys over high performance networks. In *Proceedings of SC 2002*, November 2002.

[20] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, 1997.

[21] Beth Plale. Leveraging Runtime Knowledge about Event Rates to Improve Memory Utilization in Wide Area Data Stream Filtering. In *IEEE High Performance Distributed Computing (HPDC)*, August 2002.

[22] Beth Plale and Karsten Schwan. Dynamic Querying of Streaming Data with the dQUOB System. *IEEE Transactions on Parallel and Distributed Systems*, 14(3), April 2003.

[23] Christian Poellabauer, Hasan Abbasi, and Karsten Schwan. Cooperative run-time management of adaptive applications and distributed resources. In *Proceedings of the tenth ACM international conference on Multimedia*, 2002.

[24] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications. In *Proceedings of the Conference on Principles and Practices of Parallel Programming (PPoPP)*, pages 183–192. ACM Press, May 1999.

[25] Umakishore Ramachandran, Rishiyur S. Nikhil, James M. Rehg, Yavor Angelov, Arnab Paul, Sameer Adhikari, Kenneth M. Mackenzie, Nissim Harel, and Kathleen Knobe. Stampede: A Cluster Programming Middleware for Interactive Stream-Oriented Applications. *IEEE Transactions on Parallel and Distributed Systems*, 14(11), November 2004.

[26] P.-G. Raverdy, H. L. V. Gong, and R. Lea. Dart: A reflective middleware for adaptive applications. Technical report, University of Tsukuba, 1998.

[27] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Anthony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., 2003.