2nd IEEE International Conference on Cloud Computing Technology and Science

Dynamic Resource Provisioning for Data Streaming Applications in a Cloud Environment

Smita Vijayakumar Qian Zhu Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University Columbus OH 43210
{*vijayaks,zhuq,agrawal*}*@cse.ohio-state.edu*

***Abstract*—The recent emergence of *cloud computing* is making the vision of *utility computing* realizable, i.e., computing resources and services from a cloud can be delivered, utilized, and paid for in the same fashion as utilities like water or electricity. Current cloud service providers have taken some steps towards supporting the true *pay-as-you-go* or a utility-like pricing model, and current research points towards more fine-grained allocation and pricing of resources in the future.**

In such environments, resource provisioning becomes a challenging problem, since one needs to avoid both under-provisioning (leading to application slowdown) and over-provisioning (leading to unnecessary resource costs). In this paper, we consider this problem in the context of streaming applications. In these applications, since the data is generated by external sources, the goal is to carefully allocate resources so that the processing rate can match the rate of data arrival. We have developed a solution that can handle unexpected data rates, including the transient rates. We evaluate our approach using two streaming applications in a virtualized environment.

I. Introduction

Utility computing was a vision stated more than 40 years back [18]. It refers to the desire that computing resources and services be delivered, utilized, and paid for as utilities such as water or electricity. The recent emergence of *cloud computing* is making this vision realizable. Examples of efforts in this area include Infrastructure as a Service (IaaS) providers like Amazon EC2 [1] and Software as a Service (SaaS) providers like Google AppEngine [3] and Microsoft Azure [2]. In brief, cloud comprises the available computation and storage resources over the Internet. For many small and new companies, this approach has the advantage of a low or no initial cost, as compared to having to acquire and manage hardware. In addition, a key advantage of this model is the dynamic scalability of resources and services, with a *pay-as-you-go* model, consistent with the vision of utility computing.

The *elasticity* offered by the cloud computing model avoids both under-provisioning and over- provisioning of resources, which have been typical problems with a model where a fixed set of resources were managed by a company or a user. In other words, dynamic provisioning of computing and storage resources is possible in cloud computing. With a *pay-as-you-go* model, resource provisioning should be performed carefully, to keep the resource budget to a minimum, while meeting an application's needs. Current cloud service providers have taken some steps towards supporting the true *pay-as-you-go* or a utility-like pricing model. For example, in Amazon EC2, users pay on the basis of number of type of instances they use, where an instance is characterized (and priced) on basis of parameters like CPU family/cores, memory, and disk capacity. The ongoing research in this area is pointing towards the possibility of supporting more fine-grained allocation and pricing of resources [10], [13]. Thus, we can expect cloud environments where CPU allocation in a virtual environment can be changed on-the-fly, with associated change in price for every unit of time.

In such cloud environments, resource provisioning becomes a challenging problem. In this paper, we consider this problem in the context of streaming applications. A significant development over the last few years has been the emergence of *stream* model of data (processing). 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. Two trends have contributed to the emergence of this model for scientific applications. First, scientific simulations and increasing numbers of high precision data collection instruments (e.g. sensors attached to satellites, medical imaging modalities, or environmental sensors) are generating data continuously, and at a high rate. The second is the rapid improvement in the technologies for Wide Area Networking (WAN). As a result, often the data can be transmitted faster than it can be stored or accessed from disks within a cluster. In view of the growing popularity of streaming model for data processing, a number of systems have been developed in the last 6-7 years specifically targeting this class of applications [11], [7], [6], [20], [8], [22].

Resource provisioning for streaming applications involves several unique challenges. Since the data is generated by external sources, the goal is to carefully allocate resources so as to match the rate of data arrival. A higher provisioning of resources will unnecessarily increase the resource budget. At the same time, with a lower provisioning of resources, the processing rate will fail to match the data arrival rate, and eventually cause buffer-overflow and loss of data. In our approach, each stage of distributed streaming application run on a server is monitored for its current load pattern. We want to provision the resources to match the processing time with that of data generation. This problem is trivial for expected data arrival rates. The challenge lies in tackling unexpected data rates, especially, the transient rates, which is a known characteristic of streaming applications.

We have evaluated our resource provisioning algorithm extensively for different applications under varying data arrival characteristics. The main observations from our experiments are as follows. First, the algorithm is able to converge to an optimal or near-optimal CPU allocation within a reasonable amount of time. Second, the convergence is independent of the initial resource allocation. The algorithm is capable of adapting even from extreme cases of resource over or under allocation. Finally, the algorithm is able to adapt to dynamically-varying data rates, and converges at new resource allocations in a short time frame.

The rest of the paper is organized as follows. In Section II, we present background on cloud computing and streaming algorithms and applications. The resource provisioning algorithm is described in Section III. Experimental evaluation of our algorithm is presented in Section IV. We compare our work with related research efforts in Section V and conclude in Section VI.

II. Background: Cloud Computing and Streaming Applications

This section gives a background on cloud computing environments and streaming applications. We also outline two streaming algorithms that have been used in our experimental studies.

A. Cloud Environments

We assume that the cloud computing resources are virtualized to hide their hardware and software heterogeneity and allow secure sharing between different applications and users. Furthermore, we assume that CPU cycles on a server can be effectively shared between different virtualized machines.

In our implementation and experiments, we use virtual machines (VMs) that run on top of the Xen hypervisor [19]. This system provides a Simple Earliest Deadline First (SEDF) scheduler that implements *weighted fair sharing* of the CPU capacity among all the VMs. The share of CPU cycles for a particular VM can be changed at runtime. The SEDF scheduler can operate in two modes: *capped* and *non-capped*. In the capped (or *non-work-conserving*) mode, a VM cannot use more than its share of the total CPU time in any time-interval, even if there are idle CPU cycles available. In contrast, in the non-capped (or *work-conserving*) mode, a VM can use extra CPU time beyond its share, if other VMs do not need them.

Our work particularly focuses on the *non-work-conserving* use of VMs, where multiple VMs could share the same CPU effectively, and a fair pricing model can be supported. For simplicity, we assume that the only cost associated with our target class of applications is the computing cost. Since we consider CPU cycles allocation to multiple VMs, each percentage of the CPU cycles comes with a price. We assume that the computing costs are proportional to the CPU cycles that are allocated.

B. Streaming Applications

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.

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, and, 4) it is often not feasible to store all data for processing at a later time, thereby, requiring analysis in *real-time*.

We next describe the two streaming applications used as benchmarks in our experimental study. They involve two of the most popular data mining tasks, frequent itemset mining and clustering, over streaming data. Thus, they are good representatives of the streaming data analysis class of applications.

The first application is clustering evolving data streams [5], 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 [5] we use for solving the problem is 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 *macro* clusters. After a certain number of data points have been processed in the first step, they are sent to the second processing stage. The modified k-means algorithm [5] is applied at this stage to create and output the final clusters.

The second application we have studied finds frequently occurring itemsets in a distributed data stream and is referred to as *Approx-Freq-Counts* [14]. The algorithm we use is the one proposed for finding frequent items from distributed streams [14]. The algorithm addresses the problem by defining two parameters: support s and error ϵ. 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 infrequently occurring 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.

III. Resource Provisioning Algorithm

This section describes the dynamic CPU adaptation algorithm we have developed and implemented. We will first list the main challenges for the algorithm, and then describe an application model we use in presenting the algorithm. Finally, we give a detailed description for the algorithm.

A. Key Goals

There are four main goals behind the design of our algorithm.

- Enable a given streaming application to achieve the *optimal* CPU allocation for the incoming data stream. The amount of time between receiving two consecutive units of data will then be approximately equal to the processing time for one unit. This will eliminate idle CPU time, which would be the case if there is an over-allocation. It will also avoid the possibility of data processing rate being slower, leading eventually to a buffer overflow. A related goal is to be able to converge quickly to the correct allocation, to avoid any buffer overflow, or continuing resource under-utilization.
- Support *dynamic adaptation* of CPU allocation by constant monitoring of incoming data rates. Since for streaming applications, data rates can change dynamically, we need an adaptive system, which can quickly react to a change in the application needs. At the same time, the CPU allocation must be stable if the data arrival rate remains constant. Small variation in data arrival or processing times should not lead to change in CPU allocation.
- Be capable of supporting different streaming applications with the same adaptation framework.
- The algorithm must have low overheads associated with it, so as to not slow down the overall system or application processing.

We make one assumption in our work, which is that the streaming application is compute-intensive. If the application spends dominant amount of time on communication or I/O, our algorithm will not be effective, since these overheads are not considered in decision-making.

B. Application Model

The streaming applications we consider are implemented as a set of pipelined stages. Every stage acts as a server, which processes data received from the previous stage. Every stage has an *input buffer* to queue the incoming data, and an *output buffer* to schedule transfer to the next stage in the pipeline. An example model is shown in Figure 1.

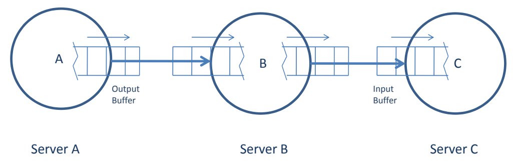


Figure 1. Queuing Model of an Application with Three Stages

Typically, the first stage is the producer of streaming data and the subsequent stages of the pipeline are the processing stages. Hence, the first stage, A, is assumed to have only an output buffer. Similarly, since the last stage does not have to transfer data to the next stage, it is assumed to have only an input buffer. All intermediate stages have both an input and an output buffer associated with them.

As we stated earlier, the main idea behind our resource provisioning algorithm is to minimize the difference in the time interval between receiving two blocks of data and the time required to process one block of data. Formally, suppose the data arrives on a given server in fixed packet sizes. Let the data arrival rate for this stage be δ and the rate of processing at this stage be μ. Since the input buffer is finite and stream application may be very long running, it is not acceptable that $\delta > \mu$. The processing rate, μ, can be increased by increasing the computing resources allocated to this node. But in the pay-as-you-go pricing model, cost is associated with the amount of computing resources. Hence, we need to allocate just the right level of CPU resources. The goal of the algorithm is to converge at an *optimal* CPU allocation such that $\delta \approx \mu$.

C. Detailed Description

Our algorithm works on the difference between the *write time* of one block of data and the time required to process this block of data. The write time is obtained by monitoring the number of bytes written into the buffer over a fixed time interval. The processing time, on the other hand, is calculated as the amount of time required to process one block of data.

The algorithm is formally shown in Figure 2. The load characteristics are averaged over a large number of samples, referred to as SAMPLE_COUNT, before they are compared. Samples are collected over a specified interval, called the MONITOR_WINDOW. After the time duration MONITOR_WINDOW, the resource provisioning manager wakes up to collect I/O and CPU utilization statistics. The data arrival during this window is recorded. Similarly, we determine the time it took for processing the last block of data. Each of these two values are then averaged over SAMPLE_COUNT.

```
ALGORITHM CPUMANAGER ()
  // Read the buffer statistics
  $w\_time = \sum_{i=0}^{SAMPLE\_COUNT} cur\_round\_writeTime$

  $p\_time = \sum_{i=0}^{SAMPLE\_COUNT} cur\_round\_processTime$

  // Determine which of coarse or fine grained adjustment required
  if  $\|p\_time - w\_time\| > MAX\_LAG \times p\_time$
    if  $(p\_time - w\_time) > 0$
      if  (buffer occupancy is in high occupancy region)
      then
        // CPU allocation is low with chance of buffer-full
        $CPU_{new} = CPU_{old} \times HIGH\_OCC\_FACTOR$
      else
        // CPU allocation is low
        $CPU_{new} = CPU_{old} \times NORMAL\_OCC\_FACTOR$
    else
      if  (buffer occupancy is in underflow region)
      then
        // CPU allocation is too high with chance of buffer-empty
        $CPU_{new} = CPU_{old} - LOW\_OCC\_DECREASE$
      else
        // CPU allocation is too high
        $CPU_{new} = CPU_{old} - NORMAL\_OCC\_DECREASE$
  elseif  $\|p\_time - w\_time\| > FINE\_LAG \times p\_time$
    if  $(p\_time - w\_time) > 0$
      // CPU allocation is a little low
      $CPU_{new} = CPU_{old} + NORMAL\_INCREASE$
    else
      // CPU allocation is a little low
      $CPU_{new} = CPU_{old} - NORMAL\_DECREASE$
  return $(CPU_{new})$
```

Figure 2. CPU Allocation Algorithm

After collecting the statistics, the resource provisioning manager examines the difference between the average processing time per block and the average time for receiving one block of data. If the processing time is larger, it indicates that data is being received much faster than the current processing capability of the node. This triggers an increase in CPU allocation. Similarly, if the processing time is smaller, it indicates that the data is being processed much faster than is being received. Hence, there are idle CPU cycles, and the manager decreases the CPU allocation.

This difference between the processing time and rate of data arrival is expressed as a fraction of the processing time. *Fine-grained* and *coarse-grained* adjustments are made based on this fraction. If the difference, measured as a fraction, is beyond a certain threshold, referred to as MAX_LAG, then a large or a *coarse-grained* adjustment is applied. Otherwise if this fraction is more than the FINE_LAG, but below MAX_LAG, then a *fine-grained* adjustment is applied. In our implementation, the value of MAX_LAG and FINE_LAG are 20% and 10%, respectively. These values ensure rapid convergence, and also ensure that CPU allocations are not varied based on small jitters in data arrival or processing rates.

The underlying idea of increase and decrease in CPU allocation is drawn from the TCP congestion control mechanism [17]. A *Multiplicative Increase* and *Additive Decrease* approach is followed by our algorithm while making coarse-grained adjustments. The reason is that it may be acceptable to have some over-provisioning, but under-provisioning is not acceptable. Consider, for example, a scenario when the data rates increase drastically after the system is in stable state. It is required that the system does not lose incoming data due to buffer overflow. In our design, rapid increase in processing rate is achieved with the use of multiplicative increase. If the CPU allocation becomes too high, then the linear adjustments will be applied to the system, so that it converges to the new optimal CPU value, and the system comes to a new stable state. Thus, the advantage of this approach is that the CPU does not stay under-allocated for too long.

The multiplicative increase in our implementation (*NORMAL_OCC_FACTOR*) is 10% over the current allocation, whereas the additive decrease for course-grained adjustments (*NORMAL_OCC_ DECREASE*) is a fixed 3%, or 3 units, of CPU. Based on the same principle, i.e., over-provisioning is preferable to under-provisioning, the fine grained adjustments are 2 units additive increase (*NORMAL_INCREASE* and 1 unit additive decrease *NORMAL_DECREASE*). These values were chosen in our implementation to achieve a tradeoff between convergence rate, while avoiding jitters, for a range of load characteristics.

A special provision is made by the algorithm for handling extreme cases of buffer occupancy. High and low occupancy cutoffs are defined as 85% and 15%, respectively, of the buffer size. If the buffer has a high occupancy and the processing time is large as compared to interval between receiving data, the coarse adjustment made (*HIGH_OCC_FACTOR*) is higher than if the buffer were in normal occupancy state (*NORMAL_OCC_FACTOR*). Similarly, the magnitude of decrease with over-provisioned CPU allocations is more in low buffer occupancy state (*LOW_OCC_DECREASE*) than otherwise (*NORMAL_OCC_DECREASE*). Fine-grained adjustments are always linear (*NORMAL_INCREASE* and *NORMAL_DECREASE*).

Consider a situation where an overload condition is detected. The CPU is increased by a fraction of the current allocation. In our implementation, it will increase to 1.1 times the current CPU allocation. An underloaded system on the other hand, will decrease CPU only by a certain fixed amount (3% in our implementation) for each iteration of the CPU management till fine-grained adjustments can be applied. The magnitudes of these adjustments applied in each direction are small in our algorithm. This is done to ensure a gradual convergence to the optimal allocation. The constants defined are not application dependent, and work well irrespective of the distributed application supported.

IV. Experimental Evaluation

This section describes the experiments we conducted to evaluate our resource provisioning algorithm. Specifically, we designed experiments with the following goals:

- To demonstrate that the CPU adaptation algorithm is able to converge to a *near-optimal* value for different static data arrival rates.
- To show that the algorithm is quickly able to converge to the new *near-optimal* value when the stream data arrival rate changes dynamically.
- To show that the algorithm is not sensitive to the initial value of CPU allocation.

A. Experimental Setup

We chose Xen as the virtualization technology and we used Xen-enabled 3.0 SMP Linux kernel in a stock Fedora 5 distribution. It is supported on multi-core servers with Intel Xeon CPU E5345, comprising two quad-core CPUs, with 8 MB L2 cache and 6 GB main memory. Each core has a clock frequency of 2.33 GHz. These hardware resources are shared between the virtual machines. Particularly, the processing node is allocated 2.5GB memory, to support the large-sized data structure without thrashing.

The CPU allocation strategy is tested against the two applications, *CluStream* and the *Approx-Freq-Counts*, which were outlined in Section II-B. The CluStream application is implemented as a three-staged processing application with the three stages being the producer, micro-cluster generator and macro-cluster generator, respectively. *Approx-Freq-Counts* is implemented as a two-stage processing application which includes a producer and a frequent-itemset generator. In both applications, our data set is the generated by the producer as a continuous stream. In the *CluStream* application, the micro-clustering stage is the most compute-intensive, whereas, in *Approx-Freq-Counts*, it is the frequent-itemset processor.

We evaluate our CPU convergence algorithm against the CPU load characteristics displayed by Xentop [4]. This utility, similar to *top*, is used to monitor memory and CPU consumptions of Xen VMs.

B. Convergence in a Static Environment

In these set of experiments, the producer produces data at a constant rate, with one block of data being generated at regular intervals. We executed the resource provisioning algorithm with static data rates, and checked for the convergence of CPU allocation to an *ideal* value with different initial values.

The first set of experiments used *CluStream*. Results in Figures 3(a) and 3(b) correspond to different computational workloads with varying initial CPU allocations. For the first case (Figure 3(a)), the ideal CPU requirement was found to be around 36% by using xentop. We consider this to be a *near-optimal* value that we will like to see convergence to. As can be seen, the algorithm is found to converge to around 36% CPU allocation with both very large and very small initial CPU values (100% and 5% respectively). In both cases, the convergence occurs after about 400 seconds. This result shows that the resource manager is able to converge to a near-optimal allocation in a reasonable time, considering the long-running nature of the streaming applications. One trend to note is that when the initial CPU allocation is very small, no change in CPU allocation is done for the first 200 seconds. This is because the CPU manager executing with a very low CPU allocation takes a longer time to accumulate the same number of statistical points (like write time and processing time averaged over a number of samples). This increases the delay in adaptation.

The results from Figure 3(b) follow a similar trend, and the ideal convergence observed for this experiment is 71%. As can be seen from the results, our algorithm predicts and converges to around 71% allocation from the higher and lower ends.

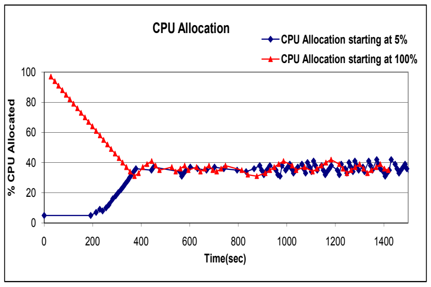
It is interesting to note the difference in the rising curve for Figures 3(a) and 3(b). This is a direct consequence of the *HIGH_OCC_FACTOR* and *NORMAL_OCC_FACTOR* (Section III-C). Since the difference between the processing time and buffer write time is large in the second case, the correction applied is *HIGH_OCC_FACTOR*. The first case sees a normal, coarse-grained difference and handles it by applying a *NORMAL_OCC_FACTOR* adjustment. This demonstrates the advantage of our proposed algorithm, i.e., it can always keep up with the data rates since provisions are made by the algorithm to react in accordance with current buffer occupancy.

We conducted the same experiments with *Approx-Freq-Counts* application, and the results are shown in Figure 4(a) and 4(b). The ideal convergence value, as reported by xentop, is 36% and 70% respectively, corresponding to different data arrival rates in these experiments.

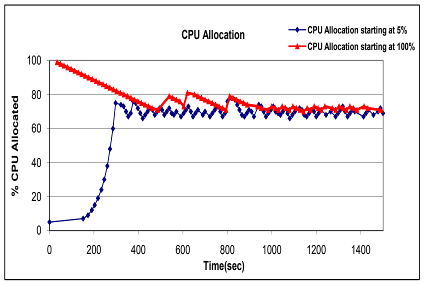
In Figure 4(b), the algorithm takes nearly 250 seconds to converge to the ideal value starting from an over-allocation, while it takes around 500 seconds starting from under-allocation. As stated earlier, this is because the CPU manager executing with a very low CPU allocation takes a longer time to obtain sufficient statistical information. This trend is more evident in the second application since the processing time is larger than the previous application. It should also be noted that a very low initial allocation is an extreme case included only to evaluate our algorithm, and is unrealistic for a compute-intensive application. Overall, our results indicate that the algorithm is able to detect over-allocation and under-allocation conditions, and the convergence is independent of the initial allocation.

One interesting point to note is that while converging from an initial under-allocation, the algorithm might do an over-allocation before converging. This is a direct consequence of the multiplicative increase, which was explained in Section III. While applying coarse-grained adjustments, the CPU allocation exceeds the optimal value. The manager then observes an over-allocation, and linearly decreases the CPU allocation, till it converges at the optimal value.

Another set of experiments demonstrate the effectiveness of our convergence algorithm when multiple nodes in the

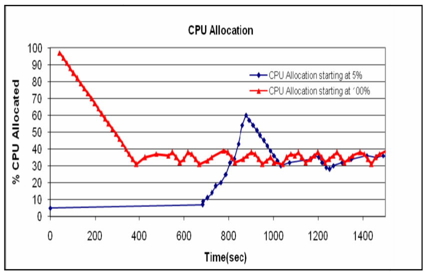


(a) Convergence to 36% CPU Allocation

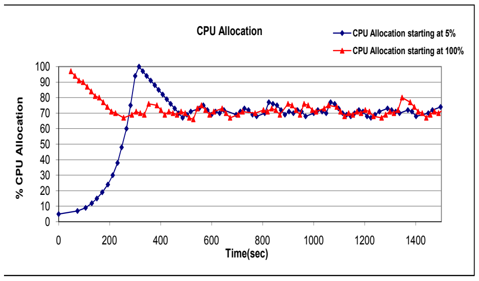


(b) Convergence to 71% CPU Allocation

Figure 3. Resource Provisioning: Convergence to Different CPU Allocations for Different Initial Allocations in *CluStream*

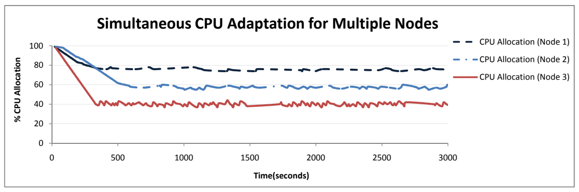


(a) Convergence to 36% CPU Allocation

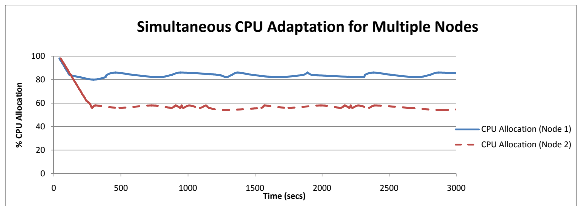


(b) Convergence to 70% CPU Allocation

Figure 4. Resource Provisioning: Convergence to Different CPU Allocations for Different Initial Allocations in *Approx-Freq-Counts*



(a) Multi-Node Convergence for *CluStream*



(b) Multi-Node Convergence for *Approx-Freq-Counts*

Figure 5. Simultaneous Convergence for Multiple Processing Nodes

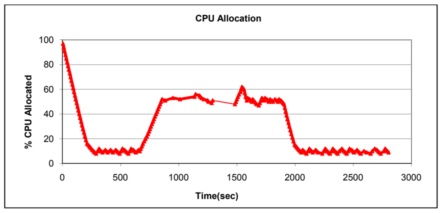
compute environment require simultaneous resource management. Figures 5(a) and 5(b) show that optimal CPU is allocated to each of the compute-intensive nodes in the system. For *CluStream*, we have three producers, with corresponding consumers processing data with different accuracy goals. These accuracy goals are captured by varying the number of micro-clusters, which are 220, 33, and 25, for these three consumers. A higher number of micro-clusters implies a larger computational workload for the consumer. We see that the algorithm converges pretty close to the Xentop values of 73%, 51%, and 34% CPU respectively for the three consumers.

In experiment with *Approx-Freq-Counts*, we considered two producers, which actually had different data production rates. They produced 4500 and 2500 samples every 5 milliseconds, or, overall rates of 3.6 and 2.0 MB per second, respectively. Figure 5(b) shows that the CPU allocation for the two corresponding consumers converges to 84% and 57% allocation, reflecting the different workload due to different production rates.

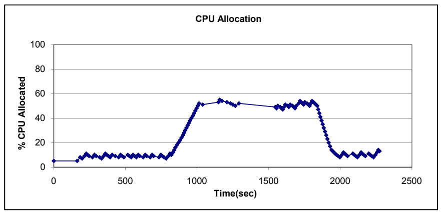
C. Convergence in a Dynamic Environment

We next run similar experiments with the two applications against dynamically varying data rates. To emulate a dynamic environment, we use two producers, $P1$ and $P2$. Individually, they have fixed data production rates. In our experiment, $P1$ is the initial producer. It is joined by $P2$ at later time, hence the net data rate is higher. Finally, when $P2$ finishes producing, the data rate falls to $P1$'s production rate.

The results from the *CluStream* application are shown in Figures 6(a) and 6(b). The ideal convergence, measured by xentop, is 15% for $P1$ alone, and 53% when $P1$ and $P2$



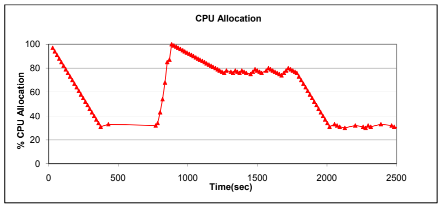
(a) Convergence from 100% Initial CPU Allocation



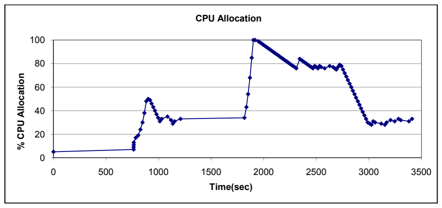
(b) Convergence from 5% Initial CPU Allocation

Figure 6. Convergence for 15% - 53% - 15% Dynamic Case in *CluStream*

are combined. Consider the first case(i.e. Figure 6(a)). The convergence to $P1$'s data rate takes around 300 seconds. After 670 seconds, $P2$ also starts producing data. The CPU manager recognizes an under-allocation and starts increasing the CPU allocation. At around 800 seconds, the CPU allocation stabilizes to around 53%. After around 1870 seconds, $P2$ stops producing data, and the CPU allocation stabilizes to 15% 100 secs later. The results captured in Figure 6(b) follow a similar trend. We observe the algorithm converges to optimal value for transient data rates irrespectively of the initial allocations (100% or 5%).



(a) Convergence from 100% Initial CPU Allocation



(b) Convergence from 5% Initial CPU Allocation

Figure 7. Convergence for 32% - 76% - 32% Dynamic Case in *Approx-Freq-Counts*

We repeated the same experiment with the second application. The ideal convergence on the system is 32% with $P1$, and 76% with both $P1$ and $P2$ producing data. The results are shown in Figure 7(a) and 7(b), corresponding to initial allocations of 100% and 5%, respectively. After the initial converge to 32% is achieved, the two graphs have similar trends. The rising trends are mostly multiplicative increase, and the falling trends are additive decreases. In some cases, CPU is over-allocated, even after the allocations are already in the optimal range. This can be explained due to the presence of intermediate carrier threads that pass data between processes (both inter-stage and intra-stage). It has been seen that by the time data rates are observed by the monitor thread, the data no longer arrives at a purely constant rate. This leads to the CPU manager concluding a temporary increase in data rate, resulting in an over-allocation, which is later corrected. Also, the *Approx-Freq-Counts* application is more compute-intensive than *CluStream*. Since high occupancy cases occur more frequently, hence, multiplicative trends are observed more often with this application than with *CluStream*.

Recall that one of the goals of our algorithm was to have very low overheads, i.e., not require many CPU cycles for monitoring the application stages and adjusting CPU allocation. In all our experiments, the resource manager did not consume more than 0.1% of the available CPU cycles. Thus, the resource manager cannot slowdown the overall application processing or add to the resource costs.

V. Related Work

We now discuss the research efforts relevant to our work from the areas of resource allocation for streaming processing and dynamic virtualized CPU provisioning.

Resource Allocation for Streaming Processing: Over the last several years, many research groups have developed distributed data stream processing systems, including SystemS [11], Borealis [7], STREAM [6], dQUOB [20], GATES [8], and Aurora and Medusa [22]. Resource allocation or provisioning is a problem considered by several of these projects. Schneider *et al.* [21] proposed an elastic scaling of data parallel operators for stream processing. They proposed an adaptive algorithm to adjust the level of parallelism at runtime such that the system can handle data burst based on the current workload on the node. Gaber *et al.* [15] adapt the data mining algorithm output on streaming applications according to resource availability and data arrival rate. Particularly, the authors considered memory as a resource constraint. Similarly, we adapted CPU allocation to match data arrival rate and processing rate during its execution. However, our work considers a different problem, i.e., we process distributed streaming applications in a cloud environment with virtualization. We have addressed the problem of dynamically changing virtualized CPU allocation to virtual domains so that there is no bottleneck in the processing chain to achieve high accuracy.

Virtualized CPU Provisioning: In last 2-3 years, the problem of virtualized CPU provisioning has received a lot of attention [16], [9], [12]. Both Palada *et al.* [16] and Kalyvianaki *et al.* [12] have applied control theory for dynamic virtualized CPU allocation for multi-tier web applications. Vitual CPU allocation is adjusted at runtime based on the variation in workloads to achieve the goal of application QoS metrics, such as throughput and response time. The distinct aspect of our work is the focus on data streaming applications, where data has to be processed in *real-time*. The goal is to adjust virtualized CPU allocation so that the processing rate can meet the data arrival rate, while optimizing the computing costs.

VI. Conclusions

This paper has considered the problem of resource provisioning for data streaming applications in virtualized or cloud environments. We have developed an algorithm which can handle dynamic patterns in data arrivals. The algorithm avoids any slowdown in processing of applications, while also conserving the resource budgets.

Our experimental study has been conducted on two popular stream data mining algorithms. The summary of our results are as follows. The resource provisioning algorithm correctly converges to the optimal CPU allocation based on the data arrival rate and computational needs. The algorithm identifies over-flow and under-flow conditions and converges to the same level, irrespective of the initial allocation. Load statistics are observed and CPU adjustments made, to arrive at the optimal allocation. The algorithm is effective even if there are significant changes in data arrival rates. The overhead of the algorithm is quite small. The overall advantage of our approach is that the system can automatically tune itself based on resource needs.

References

[1] Amazon elastic compute cloud. In *http://aws.amazon.com/ec2/.*

[2] Azure services platform. In *http://www.microsoft.com/azure/default.mspx.*

[3] Google app engine. In *http://code.google.com/appengine/.*

[4] Man page xentop. In *http://linux.die.net/man/1/xm.*

[5] 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.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The International Journal on Very Large Data Bases*, 13, 2004.

[7] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 13–24, June 2005.

[8] L. Chen, K. Reddy, and G. Agrawal. Gates: A grid-based middleware for distributed processing of data streams. In *Proceedings of the 13th IEEE Conference on High Performance Distributed Computing (HPDC04)*, pages 192–201, June 2004.

[9] S. Govindan, A. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. CPU Scheduling for Consolidated Xen-Based Hosting Platforms. In *Proceedings of Conference on Virtual Execution Environments (VEE)*, June 2007.

[10] J. Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM09)*, pages 630–637, June 2009.

[11] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOID06)*, pages 431–442, 2006.

[12] E. Kalyvianaki, T. Charalambous, and S. Hand. Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proceedings of the 6th International Conference on Autonomic Computing (ICAC09)*, pages 117–126, June 2009.

[13] H.C. Lim, S. Babu, J.S. Chase, and S.S. Parekh. Automated control in cloud computing: Challenges and opportunities. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds (ACDC09)*, pages 13–18, June 2009.

[14] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 346–357, June 2002.

[15] M.M.Gaber. Data stream mining using granularity-based approach. *Foundations of Computational*, 206(6):47–66, 2009.

[16] P. Padala, K.G. Shin, X. Zhu, M .Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys07)*, pages 289–302, Mar. 2007.

[17] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, 1998.

[18] D. Parkhill. *The Challenge of Computer Utility*. Addison-Wessley, 1966.

[19] P.Barham, B.Dragovic, K.Fraser, S.Hand, T.Harris, A.Ho, R.Neugebauer, I.Pratt, and A.Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP03)*, pages 64–177, 2003.

[20] B. Plale and K. Schwan. Dynamic querying of streaming data with the dquob system. *IEEE Transactions on Parallel and Distributed Systems*, 14, 2003.

[21] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.L. Wu. Elastic scaling of data parallel operators in stream processing. In *Proceedings of the 23rd International Parallel and Distributed Processing Symposium (IPDPS09)*, pages 1–12, May 2009.

[22] S. Zdonik, M. Stonebraker, M. Cherniack, U. Cetintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. *IEEE Data Engineering Bulletin*, 26, 2003.