Adaptive Control of Extreme-scale Stream Processing Systems

Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, Olivier Verscheure
IBM T. J. Watson Research Center, Yorktown Heights, NY

***Abstract*— Distributed stream processing systems offer a highly scalable and dynamically configurable platform for time-critical applications ranging from real-time, exploratory data mining to high performance transaction processing. Resource management for distributed stream processing systems is complicated by a number of factors – processing elements are constrained by their producer-consumer relationships, data and processing rates can be highly bursty, and traditional measures of effectiveness, such as utilization, can be misleading. In this paper, we propose a novel distributed, adaptive control algorithm that maximizes weighted throughput while ensuring stable operation in the face of highly bursty workloads. Our algorithm is designed to meet the challenges of extreme-scale stream processing systems, where overprovisioning is not an option, by making the best use of resources even when the proffered load is greater than available resources. We have implemented our algorithm in a real-world distributed stream processing system and a simulation environment. Our results show that our algorithm is not only self-stabilizing and robust to errors, but also outperforms traditional approaches over a broad range of buffer sizes, processing graphs, and burstiness types and levels[1].**

I. Introduction

The stream processing paradigm has always played a key role in time-critical systems. Traditional examples include digital signal processing systems [9], large-scale simulation platforms [7], multimedia clients and servers [12], and high resolution rendering farms [11]. More recently, distributed stream processing systems have been developed for high performance transaction processing [19] [3] and continuous queries over sensor data [8].

In today's distributed stream data processing systems, thousands of real-time streams may enter the system through a subset of the processing nodes. In such systems, hundreds of processing nodes (PNs) may be co-located, for example within a single high performance cluster, or geographically distributed over wide areas. Applications are deployed on PNs as a network of operators, or processing elements (PEs), as depicted in Figure 1. Each data stream is composed of a sequence of Stream Data Objects (SDOs), the fundamental information unit of the data stream. Each PE performs some computation on the SDOs received from its input data stream, e.g., filter, aggregate, correlate, classify, or transform. The output of this computation could alter the state of the PE, and/or produce an output SDO with the summarization of the relevant information derived from (possibly multiple) input SDOs and the current state of the PE. In order to carry out the computation, the PE uses computational resources of the PN on which it resides. These resources are finite, and are divided among the PEs residing on the node.

Distributed stream processing systems present challenging resource management goals. For example, each PE's resource utilization is constrained by PEs that are upstream and downstream of the PE in the processing graph. Further, a PEs resource consumption may be state dependent, resulting in bursty processor and network utilization throughout the system. Even developing an appropriate measure of effectiveness is difficult because the units of work (input packets) and operations (PE computations) are unequally weighted, and therefore monitoring resource utilization alone is insufficient.

In this paper, we propose *ACES: Adaptive Control for Extreme-scale Stream processing systems*, a two-tiered approach for adaptive, distributed resource control. The first tier determines the assignment of PEs to PNs. Allocations are determined through a global optimization of the weighted throughput for the processing graph, based on a expected, time-averaged input stream rates. This global optimization is performed when PEs are deployed or terminate and periodically, to support changing workload and resource availability. Second tier decisions are made in a distributed, ongoing manner. This second tier, the resource controller, jointly optimizes the input and output rates of the PE and the instantaneous processing rate of a PE, with the express goal of stabilizing the system in the presence of burstiness.

Our resource controller uses an adaptive, scalable, distributed optimization technique. Specifically, CPU and flow control for each PE is performed using only the buffer occupancy of that PE and feedback from its downstream PEs and co-located PEs. We present a closed-loop mathematical model for our solution to show that the steady-state input rate of a PE is equal to its processing rate, and each PE reaches steady-state behavior from an arbitrary starting point.

We have implemented our solution in a distributed stream processing system developed for extreme-scale data mining and in a simulator to evaluate performance in more arbitrary distributed stream processing configurations. We demonstrate that our approach outperforms traditional approaches, in terms of weighted throughput, by over 20% in the limit of small buffers and over a wide range of burstiness levels. Additionally, as weighted throughput increases, the end-to-end delay of our approach is as little as a third of traditional approaches – a significant improvement.

The remainder of this paper is structured as follows. In Section II, we review related work and highlight differences in our approach. We present a model of distributed stream processing systems in Section III, and summarize the resource management goals in Section IV. We detail our proposed approach and experimental results in Sections V - VI. In Section VII, we discuss our conclusions.

II. Background

Stream processing jobs are relatively long running and as new work is introduced into the system, the relative weights or priorities of the various jobs may change. The task of assigning weights or priorities to jobs may be performed by a human, or it may be performed by a "meta scheduler". The goal of meta schedulers generally is to assign time-averaged allocation targets based on relative importance of work submitted to a system. In comparison,

[1] An extended version of this paper can be obtained at [1].

Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)
0-7695-2540-7/06 $20.00 © 2006 IEEE

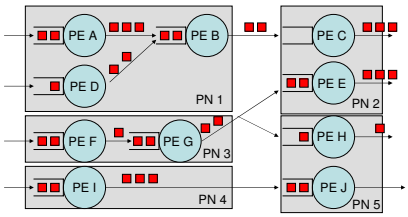

Fig. 1. Interconnection of processing elements (PEs) running on processor nodes (PNs) in a distributed stream processing system.

the goal of a resource scheduler is to enforce these long-term allocation targets.

In traditional shared processor environments, resource schedulers select a waiting process from the ready queue and allocate the resource (CPU) to it based on some assigned weight or priority. Examples include: strict or guarantee-limit enforcement [15] and velocity enforcement [5]. Neither strict or guarantee-limit enforcement consider the bursty dataflow of stream processing systems, but instead seek to enforce pre-established allocation limits. Adhering to such limits may result in input and output buffer overruns. Velocity enforcement seeks to limit how long, after requesting a resource, a PE waits for that resource. Because allocation is controlled by PE readiness for the resource, as opposed to proffered workload, the system may still be plagued by buffer overruns.

Thus, stream processing systems challenge the practice of statically assigning of priorities to PEs. To address these challenges, the River project [4] proposed effective placement of modules (PEs) and queues. However these assignments are static and determined by the application composer, as opposed to being dynamically managed during runtime. Load shedding [19] was proposed as a means to intelligently drop tuples (SDOs) from input queues, based on thresholds and potentially packet content. Dynamic placement algorithms [17] [13] have also been proposed so that operator (PE) placement can be modified according to changes in resource availability, in order to maximize some objective function on a time-averaged basis.

In each case, previous resource management work for stream processing environments targeted environments where the system must adjust operator placement or shed load in order to adapt to available underlying resources. Our work is fundamentally different in that the goal is to determine and control the resource (processor and network) allocations (placement and fractional allocations) in order to maximize an objective function and maintain overall stability. Further, instead of artificially limiting configuration changes (e.g., operator placement) [13] since such changes would destabilize the system, our work uses a control theoretic approach so the system can be self-stabilizing in the face of changes. This is especially important since changes may be induced by the scheduler or the bursty nature of the workload itself.

Our two-tiered proposal includes a meta scheduler, which assigns a fractional allocation of PN resources to each PE, and a resource scheduler that enforces the allocations in a distributed way. However, unlike conventional resource scheduling systems, our proposal takes the input data flow rate, and the *a priori* importance of the input data stream into account while allocating resources to a PE. Our resource scheduler uses only locally derived information to enforce allocations. Our approach strikes a compromise between optimality and stability by first solving for the global solution that does not take the stochastic, time-varying nature of the data flows into account, to determine nominal CPU allocations among the PEs. During run-time, these nominal allocations are adjusted to ensure stability based on the local information available to each PE. We are unaware of any such approach in the literature for the control of stream processing systems.

III. The Stream-Processing Model

This section presents an overview of the basic stream-processing model to which this work applies, and the problems that arise in controlling such networks. We discuss some issues of stream processing systems that are relevant to the allocation of resources in such systems. The simulator presented in section VI-A and the experimental system presented in section VI are both based on these basic concepts.

A. Measure of Effectiveness

To compare different mechanisms of controlling the stream processing system, we need a meaningful metric of system performance. Processor sharing algorithms typically use metrics based on resource utilization - the more work that is done, the better [14]. In the distributed stream processing context, resource utilization is not a good metric. An intermediate PE might utilize a lot of system resources, do a lot of work, and output a large number of SDOs, but that work may never make it to the output stream(s) of the system. A large allocation to such a PE may may lead to a high output rate on its output stream, but this is not productive if its downstream PEs do not have sufficient resources to process the stream into system outputs. The resource utilization metric does not capture these effects.

Network control algorithms typically use metrics based on maximizing the aggregate throughput [9], [16]. However, in a stream processing system, not all output streams contribute equally to the effectiveness of the system, i.e., the results produced by different output streams may have differing importance. The aggregate throughput metric does not capture this disparity. We would like to measure the number of SDOs the system outputs, weighted appropriately by their utility – and not just the aggregate production rate of SDOs on internal, partially processed streams. To do so, we use a *weighted throughput* metric, which attaches a positive weight to each stream that is a system output. By summing the weighted throughputs at each of these output streams, we arrive at a metric of the total productive work done by the system, in accordance with the fact that some results may be much more valuable than others. Weighted throughput has been used as a metric in other contexts in the literature [6] for reasons similar to those mentioned here.

B. Correlated Resource Usage Among PEs

PEs are constrained by their producer-consumer relationships. Most PEs in the system receive their input SDOs from other PEs, and send their output SDOs to yet other PEs for further processing. PEs cannot process SDOs at a faster (average) rate than the rate at which the upstream PE(s) produce them. Similarly, if a PE produces SDOs faster (on average) than a receiving PE can process them, the PEs will queue up in buffers until the buffers overflow. These constraints implicitly create a correlation between the resource usage of up- and down-stream PEs in a processing graph.

In addition to the correlation among up- and down-stream PEs (i.e., PEs in a single connected component), resource usage amongst PEs in separate connected components is correlated if the connected components have one or more PNs in common. Thus the effects of a resource allocation to a single PE can propagate not just through that PE's connected component, but also other connected components.

C. The Burstiness Problem

Most PEs tend to do work not in a fluid (infinitely divisible and smooth) stream of processing operations, but in relatively large chunks. For example, video processing PEs may require an entire frame, or an entire set of independently-compressed frames ("Group Of Pictures") to do a processing step. Also, consumed resources may vary according to the state of the PE or content of SDOs. Both of these factors contribute to unevenness – burstiness – in the processing rates and resource utilizations of a PE.

The classical solution to burstiness problems is to add buffers, and our stream processing system is no exception. However, designing for very high data rates and scalability in the number of PEs per processing node make buffering increasingly expensive, as system memory becomes a severe constraint. Additionally, increasing buffer sizes also increases the average end-to-end latency of the system, and decreasing latency is one of our performance objectives. So we must use what buffer space we have wisely to balance the effects of data loss, burstiness, and latency.

D. Unequal Stream Consumption Rates

A PE connected to a single downstream PE must either constrain its output rate to the input rate of the downstream PE or experience loss of SDOs when the downstream input buffer overflows. Thus, it would seem prudent to synchronize the output rate of a PE to the input rate of its downstream PE, as dropping SDOs would be wasteful (why spend processing resources creating SDOs only to drop them?).

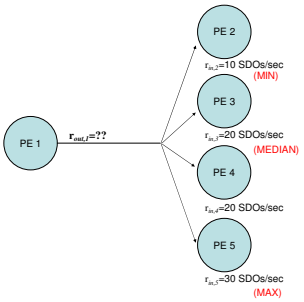

Fig. 2. Multiple PEs can read from a single stream at different rates. The sending PE can choose which of these rates at which to send data. If it chooses to send data at the rate of its fastest downstream PE (PE 5), there is buffer overflow at the other downstream PEs. If it sends data at the rate of the slowest downstream PE, the output rate of the other PEs decreases, and reduces the weighted throughput of the system.

Such a strategy, however, is not necessarily the best when a PE's output stream is read by multiple downstream PEs. Figure 2 presents an example: four PEs read the same stream, but at different rates - 10, 20, 20, and 30 SDOs/sec. For this example, assume the time-averaged CPU allocation of PE 1 is sufficient to produce SDOs at a maximum rate of 30 SDOs/sec (producing SDOs at a rate greater than this is obviously wasteful). However, using the traditional reliable delivery approach (e.g., TCP flow control) PE 1 would produce SDOs at 10 SDOs/sec – the rate supported by the slowest PE (PE 2). We refer to this policy as the *min-flow* policy. In many environments, this is indeed desirable since it prevents buffer overflows and data loss at downstream PEs. However, in a distributed stream processing system, it slows the entire PE connected component to the rate of its slowest member and thus leads to under-utilization of resources and data loss at the input to the system – and is clearly undesirable.

There are two possible approaches to solving this rate mismatch problem. The first is to *prevent* the mismatch by allocating resources such that all the downstream PEs are capable of processing incoming data at the same rate. This is often infeasible, as certain PEs may have severe resource constraints. Further, as stated in section III-A, our measure of effectiveness is the weighted throughput of the system. Since the weights of different streams vary, maximizing the weighted throughput $will$ result in unequal processing rates.

Dismissing this option, we must choose PE 1's output rate from among the different downstream PE processing rates. We argue that the maximum is the most logical choice, based on the fact that our system scheduler has chosen the resource allocations for all PEs, and we should do our best to allow them enough data to fully utilize their allocations as far as is possible. Since the system scheduler has optimized the weighted throughput, adding an additional constraint by slowing processing rates will necessarily reduce the weighted throughput. Therefore, our control algorithm attempts set the output rate of PE 1 to the input rate of its fastest downstream PE. Setting it larger than this will waste resources, and is thus sub-optimal. Setting it lower will reduce the throughput of the of fastest downstream PE of PE 1 (PE 5 in Figure 2) and necessarily reduce the weighted throughput of the system.

Thus the proposed *max-flow* policy mandates that each PE operate at its CPU allocation, and forward packets to all its downstream PEs if there is a vacancy in the input buffer of its fastest downstream PE.

IV. Control System Performance Objectives

The design and evaluation of the proposed system is motivated by a core set of design goals: maximization of weighted throughput, low end-to-end processing latency, minimization of wasted processing, and stable operation. We provided the rationale for the first 3 goals in sections III-A, III-C, and III-D, respectively. In this section, we discuss the merits of stable system operation.

A key goal of our scheduling approach is to stabilize the input, output, and processing rates of all the PEs in the system, adjusting these rates gradually over time as necessary to keep input buffers near target levels. Ensuring stable *buffer occupancy levels* has several benefits. When the system keeps enough data in incoming PE buffers, many PEs can take advantage of "batching" by processing several SDOs in quick succession (rather than having each PE process a single SDO, then executing the next PE). By batching the processing of several SDOs by the same PE, the system can avoid context-switching overhead, decrease memory cache misses, and transfer data in larger chunks (e.g. by buffering multiple output SDOs before sending them to the network). Also, when the system keeps buffers from becoming too large, end-to-end processing latency is decreased, and we avoid filling a buffer completely (which may result in upstream PEs being asked to pause their processing).

An oscillating input rate to a PE leads to an oscillating output rate and oscillatory use of processing resources. The oscillations in output rate and CPU usage can propagate through downstream and co-located PEs and destabilize the system. Also, a PE with a nearly-full buffer could force upstream PEs to oscillate between stopped (when the PE's input buffer is full) and started (when buffer space is freed). Generally speaking, local rate stability is conducive to global

stability, and thus highly desirable to create a stable distributed processing system [10].

V. ACES ALGORITHM

In this section, we present our distributed adaptive control algorithm [2]. We propose a two-tiered approach where the first tier assigns resource allocation targets to maximize the weighted throughput assuming a fluid-flow model of processing, and the second tier adjusts the instantaneous resource allocations to stabilize the system in the face of an inherently quantized and bursty workload. The first tier employs a global optimization (Section V-B) and communicates resource allocation targets to a distributed resource controller instantiated on each processing node. The second tier accepts these resource allocation targets, monitors the processing rate, input rate and quantity of buffered data for each PE, and proactively informs the distributed resource controller of the upstream PEs of the desired input rate (Sections V-C and V-D).

This decomposition is key to achieving our ambitious scalability requirements. Specifically, the first tier updates time-average resource allocations on the order of minutes and can take into account arbitrarily complex policy constraints. The second tier, which must deal with sub-second timescales involved in burstiness, is embedded in each node of the system, uses only local information and desired rate information from directly downstream PEs, and employs simple token bucket and rate tracking mechanisms.

We begin this section with review of the notation used. We then present the global optimization and provide an analysis of the steady-state and stability properties. Finally we describe the flow control and and CPU control algorithms embedded in each processing node's distributed resource controller.

A. Mathematical Preliminaries

The distributed stream processing system under consideration consists of S streams, indexed from s_0 to s_{S-1} that are inputs of the system. The system comprises of P PEs, denoted p_0, p_1, ..., p_{P-1} residing on N nodes, denoted $n_0, n_1, \ldots, n_{N-1}$. The set of all PEs and all nodes are denoted as $\mathcal{P}$ and $\mathcal{N}$, respectively. The set of PEs residing on a node is denoted as $\mathcal{N}_j$, where the subscript j denotes the node index.

The interconnection of the PEs is represented by a directed acyclic graph (DAG). We denote the set of PEs that feed data to PE j as $\mathcal{U}(p_j)$, and the set of PEs that PE j feeds data to as $\mathcal{D}(p_j)$. Thus, $\mathcal{U}(p_j)$ denotes the "upstream" PEs of p_j, while $\mathcal{D}(p_j)$ denotes the "downstream" PEs of p_j. Since the PEs at the egress of the system do not have any downstream PEs, $\mathcal{D}(p_j) = null$ for p_j at the egress. In addition, the PEs at the ingress of the system derive their input from a data stream, thus, we denote $\mathcal{U}(p_j) = s_k$ if PE p_j derives its data from stream s_k.

We discretize time by sampling in intervals of Δt and all quantities are measured at the sampled times. Denote by $r_{in,j}(n)$ and $r_{out,j}(n)$ the input and output bytes of data for PE j in the time interval $[n\Delta t, (n+1)\Delta t)$. The CPU allocation of PE p_j in the interval $[n\Delta t, (n+1)\Delta t)$ is denoted as $c_j(n)\Delta t$. The CPU allocations are represented in normalized form, thus

$$\sum_{j \in \mathcal{N}_i} c_j(n) \leq 1 \ \forall \ n \geq 0. \tag{1}$$

[2] A detailed development of the algorithm is presented in [1], with the necessary proofs.

We define $\overline{r}_{in,j}$, $\overline{r}_{out,j}$ and $\overline{c}_j$ as the time averaged values of $r_{in,j}(n)$, $r_{out,j}(n)$ and $c_j(n)$. Thus,

$$\begin{array}{r} \overline{r}_{in,j} = \lim_{n\to\infty} \frac{1}{N} \sum_{n=0}^{N} r_{in,j}(n), \\ \overline{r}_{out,j} = \lim_{n\to\infty} \frac{1}{N} \sum_{n=0}^{N} r_{out,j}(n), \\ \overline{c}_j = \lim_{n\to\infty} \frac{1}{N} \sum_{n=0}^{N} c_j(n) \end{array} \tag{2}$$

B. Global Optimization

The global optimization determines the time-averaged allocations $\overline{r}_{in,j}$, $\overline{r}_{out,j}$ and $\overline{c}_j$ for each PE such that the weighted throughput is maximized. During operation, we use a control algorithm to alter $r_{in,j}(n)$, $r_{out,j}(n)$ and $c_j(n)$ to achieve two objectives: (1) stability of the system, (2) ensure that $r_{in,j}(n)$, $r_{out,j}(n)$ and $c_j(n)$ are varied such that over a reasonably long epoch, Equation 2 is met. We refer to $\overline{c}_j$ as the long-term CPU target, and $c_j(n)$ as the CPU allocation at time $n\Delta t$.

The global optimization maximizes an aggregate utility function. Associate with PE p_j a utility $U_j(\overline{r}_{out,j})$, if its time-averaged output rate is set to $\overline{r}_{out,j}$. The function $U_j(\overline{r}_{out,j})$ is strictly increasing, concave, differentiable. We parameterize the utility function of the various PEs as $U_j(\overline{r}_{out,j}) = w_j U(\overline{r}_{out,j})$, where w_j is the "weight" of a PE (a larger weight implies higher utility), and the function $U(x)$ is is identical for all the PEs . For example, we could set $U(x) = 1 - e^{-x}$; $U(x) = log(x+1)$; $U(x) = x$. The weights $\{w_j\}$ measure the relative importance of the PEs. The cumulative utility of the system (denoted U_S) is then given as the sum of the utilities of the PEs

$$U_S(\overline{r}_{out,0}, \overline{r}_{out,1}, \ldots, \overline{r}_{out,P-1}) = \sum_{j \in \mathcal{P}} w_j U(\overline{r}_{out,j}). \tag{3}$$

We maximize Equation 3 under the following set of constraints:

$$\Sigma_{j \in \mathcal{N}_i} \overline{c}_j \leq 1 \ \ for \ 0 \leq i \leq N-1, \tag{4}$$

$$\overline{r}_{in,j} \leq \overline{r}_{out,i} \ \ for \ i \in \mathbf{U}(p_j), \ 0 \leq j \leq P-1, \tag{5}$$

$$\overline{r}_{in,j} = h_j(\overline{c}_j), \tag{6}$$

where $h_j(\overline{c}_j)$ denotes the average input rate when the CPU allocation for PE j is $\overline{c}_j$[3]. Equation 4 ensures that the CPU allocations of all the PEs on a node sum to less than one. Equation 5 ensures that the output rate of a PE is not less than the input rate of its downstream PE (the inequality in Equation 5, as opposed to an equality, stems from the fact that we enforce a max-flow policy). Lastly, Equation 6 maps the CPU allocations to the time-averaged input rates $\overline{r}_{in,j}$.

We use Lagrange multipliers to maximize Equation 3. As such any concave optimization algorithm can be used. The concavity of the cumulative utility ensures that there exists a unique set of CPU allocations $\overline{c}_j$ that maximize Equation 3.

C. Flow control algorithm

The arguments of Section III-C show that a myopic optimization strategy, where each PE processes data agnostic to the state of its co-located and downstream PEs leads to instability of the system. We propose a joint flow and CPU control algorithm to meet the stability

[3] The function $h_j(\overline{c}_j)$ is modeled as $a\overline{c}_j - b$, where a and b are constants that are determined empirically. The constant b represents the overhead involved in setting up the data structures of the PE, the overhead in function calls etc., while the constant a represents the number of bytes of input data that can be processed by the PE per processing cycle.

objectives stated in Section IV. For PE j at time $n\Delta t$, the control algorithm jointly determines $r_{in,j}(n)$, $r_{out,j}(n)$ and $c_j(n)$ in a distributed manner, taking into account the input buffer occupancy of PE j and the feedback it receives from its downstream and co-located PEs. The goal of the allocation is to maintain stability of the system, and avoid loss of partially processed data due to buffer overflow.

The distributed allocation algorithm proceeds as follows. Every Δt seconds, each PE determines its maximum sustainable input rate, based on its input buffer occupancy and its processing rate, and the maximimum sustainable rates of its downstream PEs. The PE then computes a maximum input rate using this information, and communicates this rate to its upstream PEs, and so on and so forth. Our goal is to determine this rate judiciously. A conservative choice will lead to wastage of processing resources, and an optimistic choice would cause buffer overflow. Additionally, it is desirable to have a computationally light algorithm to achieve this goal.

Towards this end, we propose an Linear Quadratic Controller (LQR), that guarantees asymptotic stability. The details of the algorithm are given in Appendix A. The optimization yields the following control equation for the maximum output rate of a PE

$$\begin{array}{c} r_{max,j}(n) = [\ \ \rho_j(n) - \sum_{k=0}^{K} \lambda_k \{b_j(n-k) - b_0\} \\ - \sum_{l=1}^{L} \mu_l \{r_{max,j}(n-l) - \rho_j(n-l)\}\ \ ]^+, \end{array} \quad (7)$$

where the subscript j indexes the PE, $\rho_j(n)$ is the processing rate of the PE in time step n and b_0, $\{\lambda_k\}$, $\{\mu_j\}$ are constants determined $a-priori$ through the approach presented in Appendix A.

The parameter b_0 is the buffer occupancy level that the controller tries to maintain. It is chosen to satisfy two objectives: (a) it is small enough to minimize the queuing delay, and avoid buffer overflow, and (b) large enough to ensure high utilization of the PE, or equivalently, minimize the chance of a buffer underflow. For a given b_0, if constants $\{\lambda_k\}$ are large (relative to $\{\mu_l\}$), the PE tries to make $b_j(n)$ equal to b_0. On the other hand, if $\{\mu_l\}$ are large relative to $\{\lambda_k\}$, the PE attempts to equalize the input and the processing rates. Next, we propose our approach to determining the CPU allocations, $\rho_j(n)$.

D. CPU Control

The CPU scheduling algorithm, which is run independently on each node, partitions the computational resources of the node among the PEs running on it based on (a) the long-term averaged CPU targets of the PEs, (b) the input buffer occupancies of these PEs, and (c) feedback from downstream PEs.

The feedback from downstream PEs provides an upper bound for the CPU allocation to a PE. At time $n\Delta t$, PE j receives an update of $r_{max,i}(n)$ from all its downstream PEs $i \in \mathcal{D}(p_j)$. PE j determines an upper bound on its output rate using this information as

$$r_{o,j}(n) \leq \max\{r_{max,i}(n) : i \in \mathcal{D}(p_j)\} \quad (8)$$

This bounds its CPU allocation $c_j(n) \leq g_j^{-1}(r_{o,j}(n))$, and consequently, its processing rate ρ_j. Note that Equation 8 embodies the max-flow paradigm discussed in Section III-D.

The allocation of CPU resources is achieved through the use of a token-bucket mechanism, where each PE running on a node earns tokens at a fixed rate, and expends them when it does processing. If a PE does not use its tokens for a period of time, it accumulates these tokens up to a maximum value. The PEs are allowed to expend their token for CPU cycles proportional to their input buffer occupancies, such that $c_j(n)$ does not exceed the bound of Equation 8. In this manner, the long-term CPU allocation of a PE on a node is maintained at its CPU target, since it accumulates tokens at a rate equal to its CPU goal. The instantaneous CPU allocation of the PEs is, however, dependent on its congestion level (i.e., buffer occupancy) and the feedback from its downstream PEs. The CPU control algorithm thus aims to mitigate congestion and loss of partially processed data while maintaining the long-term CPU targets of the PEs.

E. Summary

Our control algorithm works by backward induction on the DAG of the PEs. At time $n\Delta t$, the CPU allocation of all the PEs on a node are determined based on their buffer occupancies at time $n\Delta t$ and the number of tokens that they have accumulated – under the constraint of Equation 8. This CPU allocation is used to determine the maximum input bytes $r_{max,j}(n)$ that each PE can admit in the interval $[n\Delta t, (n+1)\Delta t)$ using Equation 7. This information is then propagated to the upstream PEs, which perform the same computation, and so on and so forth. Stability is guaranteed through the LQR equations, and asymptotic convergence to the desired state is guaranteed through the steady-state analysis (detailed in [1].

Lastly, even though we have discretized time in intervals of Δt, the algorithm does not depend on synchronization among the various nodes. In practice, the computations are performed periodically at each node, taking into account the most recent updates on the maximum input rates received by the PEs.

VI. Empirical evaluation

In this section, we evaluate the performance of ACES in a real-world distributed stream processing system, using workloads developed to model real-world conditions. We compare our proposal (denoted as the *ACES approach* (*System 1*) to two traditional approaches:

System 2: UDP: Each PE communicates SDOs to its downstream PE regardless of the input buffer occupancy of the downstream PE. If the input buffer of the downstream PE is full, the incoming SDO is dropped.

System 3: Lock-Step : Each PE communicates SDOs to its downstream PE only if the input buffer of the receiving PE is not full, i.e., the min-flow policy. If the downstream PEs input buffer is full, the upstream PE sleeps until buffer space is available. While a PE sleeps, the CPU is redistributed among the other PEs residing on the node. The long-term CPU targets of the PEs are met.

We use our primary objectives – maximize weighted throughput, minimize end-to-end latency, and maintain stability in the presence of bursty workloads – to compare our system to traditional approaches. For example, supporting a large number of high volume data streams results in contention for input buffer space. We evaluate the performance, in terms of weighted throughput and end-to-end latency, over a range of buffer sizes and show that ACES outperforms traditional approaches for both measures.

Secondly, we anticipate a range of burstiness in both network and processing resource consumption. In Section VI-C.1, we evaluated the performance, again in terms of weighted throughput, and found that, except for very low levels of burstiness, our proposal outperforms traditional approaches.

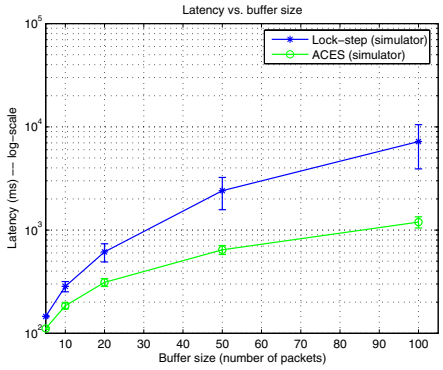

Fig. 3. Graph plotting the mean and first standard deviation in the end to end latency of the *ACES approach* and the Lock-Step .

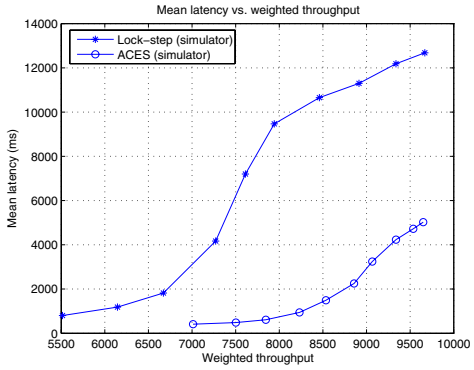

Fig. 4. Graph plotting the mean latency versus weighted throughput for the *ACES approach* and the Lock-Step in the simulator (200 PE, 80 node topology). The *ACES approach* has a lower mean latency for the same weighted throughput as the Lock-Step .

A. Simulation Environment

The simulation environment was implemented in C using the C-SIM [18] library. The physical system was replicated in the simulation environment. The simulation environment was calibrated to perform as closely as possible to the SPC system. The topologies for the simulation were generated through a topology generation tool that takes as input the number of CPUs in the system, the number of ingress, egress and intermediate PEs in the system, and the average degree of interconnectivity between the PEs. The output of the generator is a PE graph, the assignment of the PEs to the CPUs, the time-averaged CPU allocations of the PEs and the parameters for each PE.

B. PE model

We model each PE as a state-machine and characterize it through its input-output relationship. For every SDO that a PE reads from its input buffer, it processes the SDO for time T, and subsequently forwards M SDOs to its downstream PE. The PE operates in two states, $S \in \{0, 1\}$. The processing time of a packet differ in the two states, and this leads to burstiness in processing. The duration that a PE spends in state S is chosen from a continuous-time exponential distribution with parameter λ_S. A large value of λ_S signifies that the PE switches between its processing states infrequently, and vice-versa.

C. Experimental results

Experiments were run on topologies consisting of 60 PEs running on 10 nodes in the SPC and the C-SIM simulator. This was done to calibrate the simulator to the SPC. Subsequently, experiments were run on the simulator on topologies of 200 PEs on running on 80 nodes. Multiple randomly generated topologies were used for all simulations, and the results were averaged over the multiple runs. Unless otherwise stated, the buffer size of each PE was set to $B = 50$ SDOs, the parameter b_0 was set to $B/2$ SDOs, the maximum allowable fan-out degree was set to 4, the maximum allowable fan-in degree was set to 3, the fraction of PEs that had multiple inputs or multiple outputs was set to 20% and the parameters of the PEs were set to $\lambda_S = 10$, $\lambda_m = 1$, $\rho = 0.5$, $T_0 = 2ms$ and $T_1 = 20ms$.

Figure 4 plots mean latency of the *ACES approach* and Lock-Step versus weighted throughput. The variation in latency and weighted throughput was accomplished by altering the input buffer size (B) of the PEs. As is seen in the figure, the proposed approach has a superior trade-off between throughput and latency as compared to the Lock-Step approach. This is because the proposed approach maintains each PE's buffer less than full, thus packets spend less time in each buffer. In addition, the proposed approach drops fewer packets as compared to the Lock-Step approach, this leads to a higher end-to-end throughput.

1) Performance measurements with variation in burstiness: We plot the weighted throughput of the three approaches with variation in burstiness in this section. The burstiness was varied by varying the mean time the PEs spend in each of the two states before transition (Figure 5).

Overall, an increase in λ_s increases the burstiness in processing and results in a decline in the performance of the three systems. However, the *ACES approach* performs better than either of the other approaches for two reasons: (1) the proposed dynamic flow and CPU control algorithms stabilize the system such that in the short-term resources are allocated among the PEs that require them, while in the long-term a fair allocation is maintained, and (2) the merits of the max-flow policy over the min-flow policy. The figure also shows the results of the calibration of the simulator to the SPC.

VII. Conclusions

In this paper, we have proposed an adaptive control algorithm for resource sharing in a distributed stream processing environment. We outlined the characteristics of a stream processing system relevant to the problem of resource allocation – correlated usage among PEs, burstiness in processing and unequal stream consumption rates – and proposed the weighted throughput and end-to-end latency as appropriate metrics for evaluating the performance of the system.

We proposed a two-tier optimization approach. The first tier, a global optimization algorithm, determines the time averaged CPU targets of the PEs to maximize the weighted throughput, taking the correlation among the loads of the PEs into account. The second tier distributed flow control and CPU control algorithms adjust the flow of data and CPU allocations of the PEs, respectively, in real-time to ensure stability and maintain the long-term CPU targets of the PEs. Together, the two-tiers strive to maximize the weighted throughput over suitably long epochs, while maintaining stability.

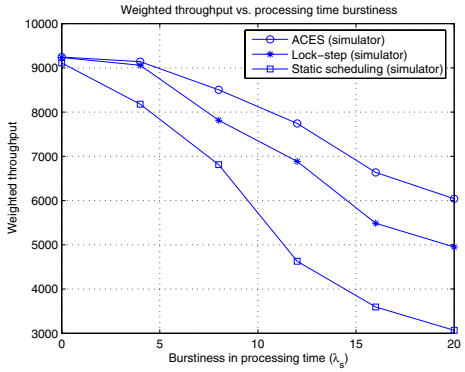

Fig. 5. Graph plotting the weighted throughput with variation in the average time spent in each of the two states of the PEs (λ_s). An increase in λ_s leads to a lower weighted throughput of the three approaches, however, the loss in performance is smaller for the *ACES approach* as compared to the other approaches.

The proposed flow control algorithm was designed using the LQR method – a robust and provably convergent design method.

Multiple experiments were conducted on the real-world SPC system [2] and a simulator calibrated to the SPC, comparing the performance of the weighted throughput and end-to-end latency of ACES to Lock-Step and UDPapproaches. It was observed that the weighted throughput achieved by ACES exceeds the other approaches, and increase as burstiness is increased. It was also shown that the performance of ACES dominates the other approaches across the entire range of operation. For large weighted throughput, it was observed that the mean end-to-end latency of ACES was as little as a third of the Lock-Step – a significant improvement. Moreover, the standard deviation of the mean end-to-end latency of ACES was much smaller than the *Lock-step approach*. The robustness of ACES to errors in allocation was also demonstrated.

In summary, it was shown that ACES outperforms conventional scheduling approaches across the relevant range of operation of the PEs, both, in terms of weighted throughput and end-to-end latency, with variation in buffer size, burstiness and errors in allocation.

Acknowledgements

ACES and the SPC are part of System S, an extreme-scale data mining system being developed as a collaborative effort among many groups within IBM Research. The authors wish to thank Nagui Halim, the principal investigator and chief architect for System S, and other System S team members for many formative discussions.

References

[1] L. Amini, N. Jain, A. Sehgal, J Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *IBM Research Report, available at www.research.ibm.com/people/a/aminil/publications.html*, Yorktown Heights, NY, January 2006.

[2] Lisa Amini, Henrique Andrade, Frank Eskessen, Richard King, Phillippe Selo, Yoon Park, and Chitra Venkatramani. The Stream Processing Core. In *under review*, 2005.

[3] Aravind Arasu, Brian Babcock, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM International Conference on Management of Data (SIGMOD 2003)*, San Diego, CA, June 2003.

[4] Remzi Arpaci-Dusseau, Eric Anderson, Noah Treuhat, David Culler, Joseph Hellerstein, David Patterson, and Kathy Yelick. Cluster i/o with river: Making the fast case common. In *Proceedings of the In Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, May 1999.

[5] P. Bari, C. Covill, K. Majewski, C. Perzl, M. Radford, K. Satoh, D. Tonelli, and L. Winkelbauer. IBM Enterprise Workload Manager. In *www.redbooks.ibm.com/*.

[6] Y. Bartal, F. Chin, M. Chrobak, S. Fung, W. Jawor, R. Lavi, J. Sgall, and T. Tichy. Online competitive algorithms for maximizing weighted throughput of unit jobs. In Proc. of the 21st Symposium on Theoretical Aspects of Computer Science (STACS'04), 2004.

[7] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: a platform for heterogeneous simulation and prototyping. In *Proceedings of the 1991 European Simulation Conference*, Copenhagen, Denmark, June 1991.

[8] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, , and Mehul Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the 2003 Conference on Innovative Data Systems Reseearch (CIDR 2003)*, Asilomar, CA, 2003.

[9] P. D. Hoang and J. M. Rabaey. Scheduling of DSP programs onto multiprocessors for maximum throughput. *IEEE Transactions on Signal Processing*, 41(6):2225–2235, June 1993.

[10] C. V. Hollot, Vishal Misra, Donald F. Towsley, and Weibo Gong. On designing improved controllers for AQM routers supporting TCP flows. In *INFOCOM*, pages 1726–1734, 2001.

[11] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Transactions on Graphics*, 2002.

[12] Rainer Koster, Andrew Black, Jie Huang, Jonathan Walpole, and Calton Pu. Infopipes for composing distributed information flows. In *Proceedings of the 2001 ACM Multimedia Workshop on Multimedia Middleware*, Ottawa, Canada, October 2001.

[13] Vibhore Kumar, Brian Cooper, and Karsten Schwan. Distributed stream management using utility-driven self-adaptive middleware. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC)*, April 2005.

[14] Tak-Wah Lam, Tsuen-Wan Ngan, and Ker-Keung To. On the speed requirement for optimal deadline scheduling in overloaded systems. In *Proc. 15th International Parallel and Distributed Processing Symposium*, page 202, 2001.

[15] Shailabh Nagar, Rik van Riel, Hubertus Franke, Chandra Seetharaman, Vivek Kashyap, and Haoqiang Zheng. Improving Linux resource control using CKRM. In *Proceedings of the 2004 Ottawa Linux Symposium*, Ottawa, Canada, July 2004.

[16] Ilkyu Park, Youngseok Lee, and Yanghee Choi. Stable load control with load prediction in multipath packet forwarding. In *ICOIN*, pages 437–444, 2001.

[17] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Networking meets databases. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, April 2006.

[18] Herb Schwetman. CSIM: a C-based process-oriented simulation language. In *WSC '86: Proceedings of the 18th conference on Winter simulation*, pages 387–396, New York, NY, USA, 1986. ACM Press.

[19] Stan Zdonik, Michael Stonebraker, Mitch Cherniak, Ugur Cetintemel, Magdalena Balazinska, and Hari Balakrishnan. The Aurora and Medusa projects. *Bulletin of the IEEE Technical Committee on Data Engineering*, March 2003.