RC23922 (W0603-160) March 24, 2006
Computer Science

IBM Research Report

Autonomic Management of Stream Processing Applications via Adaptive Bandwidth Control

Dimitrios Pendarakis, Jeremy Silber, Laura Wynter
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Autonomic Management of Stream Processing Applications via Adaptive Bandwidth Control

Dimitrios Pendarakis* Jeremy Silber Laura Wynter
IBM Watson Research Center
Hawthorne, NY, USA
{dimitris, jsilber, lwynter}@us.ibm.com

Keywords: Stream Processing, Autonomic Control, Distributed Processing,
System Management, Nonlinear Optimization, Bandwidth Control, Resource Allocation
Technical Area(s): Networking, Operating Systems, Quality of Service

Abstract

We present a novel autonomic control system for high performance stream processing systems. The system uses bandwidth controls on incoming or outgoing streams to achieve a desired resource utilization balance among a set of concurrently executing stream processing tasks. We show that CPU prioritization and allocation mechanisms in schedulers and virtual machine managers are not sufficient to control such I/O-centric applications, and present an autonomic bandwidth control system that adaptively adjusts incoming and outgoing traffic rates to achieve system management goals. The system dynamically learns the bandwidth rate necessary to meet the system management goals using stochastic nonlinear optimization, and detects changes in the stream processing applications that require bandwidth adjustment. Our prototype Linux implementation is lightweight, has low overhead, and is capable of effectively managing stream processing applications.

1 Introduction

Effective operation of modern, large-scale parallel and distributed systems critically depends on efficient management of multiple resources, such as processing cycles, link bandwidth and system memory, allocated among multiple (possibly parallelized) applications. The challenges in achieving this are amplified by two trends: First, new service models such as *On-Demand*, or *utility computing* introduce bursty and unpredictable demand patterns, which are difficult to characterize in advance. Second, successful operation of these service models critically depends on the ability to meet service level agreements on performance. However, such performance requirements are typically met by allocating system resources according to the expected resource demands of the processes.

*Corresponding author; dimitris@us.ibm.com

While substantial prior work exists on independently managing different types of resources, such as processor capacity and network bandwidth, little attention has been given to the complex dependencies between availability and utilization of each one of these types of resources and application performance. Often the approach to resource management in high performance computing and grid clusters is to size all resources so as to ensure that no bottlenecks occur across all processes. Alternatively, one of the resources, considered the more likely bottleneck, is explicitly controlled and it implicitly dictates the utilization of the other resources. For example, in systems with explicit control of processing cycle allocation, processor allocation dictates network bandwidth usage. Typically, interconnection bandwidth is overprovisioned to the point that the network can keep pace with the processor's demands for more data, and processing power is usually the bottleneck that limits throughput. However, the advent of data-intensive stream processing applications, as well as the expansion of distributed systems to global-scale (for example in grid computing), renders such overprovisioning impractical and/or prohibitively expensive. In such cases network bandwidth must be considered jointly with processing power during resource allocation.

In this paper we introduce a novel scheme for managing high performance distributed systems. In contrast to typical control mechanisms that rely exclusively on independent allocation of resources, such as processing cycles or network bandwidth, we propose using adaptive joint allocation of multiple resource driven by performance objectives. That is, instead of developing separate goals and control mechanisms for managing different resources, we propose a new paradigm for monitoring and achieving application processing objectives through explicit network controls. This approach is particularly valuable when the network becomes a constrained resource. Traditional network resource allocation schemes require system administrators and/or application developers to provide a characterization of the traffic generated by the applications and the corresponding network bandwidth requirements. Our scheme eliminates such manual intervention by autonomously developing a model that expresses the dependency between different resources. We develop a system management mechanism that autonomously monitors, for multiple simultaneously running applications, the performance relative to quality objectives, and utilization of system resources. The system learns the implicit relationship between the two and uses it to effectively manage the applications.

Specifically, we present a system for autonomic management of *network resources* (such as local link bandwidth) to effect a desired balance between concurrently executing processes on a stream processing node. This system is capable of maintaining this balance with or without the presence of explicit per-process CPU allocation mechanisms (e.g. processor sharing between VMWare [8] images, or CPU-allocating schedulers such as the IBM HyperVisor [17] or CKRM [5]). In systems without per-process CPU allocation mechanisms, such as a stock Linux kernel, our management system is able to autonomously determine the appropriate data rate to/from each process necessary to effect the desired per-process CPU usage and allocate the corresponding amount of network bandwidth. Even in systems equipped with per-process CPU allocation mechanisms, some processes may, when left uncontrolled, consume larger amounts of bandwidth, thus "'starving"' other (potentially more important) processes to the point where they cannot utilize their allocated CPU share. Our autonomic traffic management system prevents this from occurring.

Our scheme makes a number of important contributions. First, we introduce a very low-cost *dynamic learning* procedure to determine the relation between performance objectives and various types of allo-

cated resources, in particular between communication resources and processing resource. By dynamically learning this relationship through non-perturbative observation of the system, our scheme enables *autonomic* system operation; the system operator or application supplier need not explicitly specify or predict this complex, and often unknown, relationship. Second, we develop a novel stochastic Newton-type optimization algorithm that robustly and rapidly drives the system towards a desirable operating point through bandwidth control. The learning and optimization procedures are performed jointly; thus maximizing the convergence rate. In addition to its rapid convergence rate, our algorithm scheme is designed to provide both *stability* and *adaptability* in the presence of incomplete information or noise. For example, measurements of performance metrics and/or resource utilization may include substantial noise due a variety of background system operation factors. We have devised methods to filter out the impact of noise and avoid unecessary, transient, changes to the system operating state.

This paper is organized as follows: Section 2 presents an overview of related work. Section 3 presents the problem formulation and model. In section 4 we summarize the stochastic Newton-type algorithm that simultaneously learns the complex relation between two types of resources and optimizes a target-level criterion, and we provide a convergence proof. Section 5 discusses the architecture and components of the system we built using this alogrithm, followed by experimental results and comparison with other approaches in Section 6. We conclude in Section 7 with discussion and some future research directions.

2 Overview of Related Work

There is a substantial amount of prior work on the areas of independently managing networking and processing resources in distributed systems, but considerably less in exploring the dependency between these two types of resources.

The problem of providing network QoS via bandwidth reservation, scheduling and policing has been extensively researched over the past years. The main focus of this work [7, 13, 11, 15] has been to study mechanisms that, given an appropriate traffic characterization, can achieve high level of statistical multiplexing efficiency while guaranteeing the requested levels of QoS (or limiting violations thereof). The traffic parameters (such as leaky bucket rate and size) along with the negotiated QoS can be thought of as a *contract* between the user and the network. This work underlined part of the work of the Internet Engineering Task Force *Integrated Services* (IntServ) [3, 6] and *Differentiated Services* (DiffServ) [18] working groups. One common shortcoming of these approaches is that they rely on accurate characterization of the traffic streams entering the network. While this assumption might be true for a small set of well known applications, it is clearly not applicable in today's heterogenous, on demand computing envirornment, where the traffic generated by an application and its relation to CPU requirements are not known in advance. Additionally, this work does not link the allocation of network bandwidth with CPU processing utilization. Still, the techniques developed to allocate and police bandwidth allocation can be used as a building block in our control system.

On the operating systems domain a substantial amount of work in the area of managing distributed applications has been published. This work typically focuses on intelligent scheduling of different tasks across different nodes, prioritization of tasks, and allocation of CPU, memory, and disk resources. Typically

the network is assumed to be of sufficient capacity that it can be considered infinite. The authors in [12, 20] describe a feedback-control based resource manager to allocate system resources based on the measured progress of applications. Progress is measured by monitoring the occupancy level of application buffers and the heuristic target is to maintain occupancy at a desired level. Load balancing [14] has been also used to asign different nodes in a cluster to different processes requiring processing. It relies on individual nodes that periodically send state (processing load and resource availability) measurements to a gateway node that then determines the appropriate job dispatch policy. In [21] the allocation of communication resources is considered in a linear system that yields optimal system performance. In particular, the authors consider the joint problem of resource allocation and system design in order to optimize system performance.

Prior research in multimedia and stream processing systems is also relevant to our work; multimedia applications have often been developed with built-in adaptation mechanisms to handle network or system congestion. In [10] application-level quality adaptation techniques are presented. In [19] adaptive mechanisms for real-time applications are described which adjust resource allocation if there is risk of failure to satisfy timing constraints. Transcoding of web page multimedia objects based on the available bandwidth is used in [4] to provide service differentiation across different clients. Abeni and Buttazzo in [1] propose a framework for dynamically allocating CPU resources to tasks whose execution times are not known apriori. The motivation for *learning* the tasks' CPU requirements is similar to our work; however the authors consider only requirements in terms of CPU time and not network bandwidth, and do not address any dependencies between the utilization of multiple resources. A multiple-resource utilization prediction model, based on autocorrelation and cross correlation between two resources (e.g., CPU and memory) is presented in [16]. That work presents a novel model for predicting the joint utilization requirements of different resources but does not address how to achieve a desired operating point from a system management perspective.

Among more recent system management tools, VMware [8], creates virtual machines (VMs) on x86 architecture systems. It partitions a single physical system into logical compartments, each running its own copy of operating system and applications and, thus, giving the illusion of a separate system. While such separation is an effective way for managing different applications' requirements, the granularity of the different VMs is quite coarse and not very well suited for individual stream processing applications. In addition, creating and managing many different VMs and operating system images entails substantial overhead: of the order of 2-20%, depending on the application, product and experimental setting [8], substantially more than the 0.2-0.5% overhead that our system exhibits.

3 Problem Formulation

Consider n applications running on a system node, each corresponding to a process which uses (local) link bandwidth to send to and receive data from one or more remote nodes. We make the following assumption:

Assumption 1 *The set of processes, $i = 1 \dots n$ remains constant throughout a time epoch of given length, τ.*

In other words, while the system we consider is stochastic, its behavior has some minimal degree of stationarity, enough to allow our algorithm to learn the relationship between processing and network resource requirements and drive the system to the target operating point. In practice, as observed in our experimental prototype, it is sufficient for this pseudo-stationarity time epoch, τ, to be on the order of 30 seconds. The system management goal that we focus on is that of achieving a set of desired processing capacity (CPU) allocations among different processes. This goal is akin to what workload managers or load balancers try to achieve in distributed systems. In addition, this goal and the allocation of CPU resources in general are of importance to virtualization engines, such as HyperVisor, and VMWare. The method we propose could, however, be used with other system resources, such as memory, and with system management goals other than trying to achieve a particular target value.

With respect to this particular goal, we shall assume a set of *desired* processing capacity (CPU) allocations, for the n processes, $t_1, t_2, \ldots, t_n$ is provided, where t_i denotes the percentage of processing resources allocated to process i. We assume that the input target CPU levels, t_i, $i = 1 \ldots n$, satisfy $\sum_{i=1\ldots n} t_i \leq 1$.

We also denote the *observed* allocation of CPU resources to the n processes by $c_1, c_2, \ldots, c_n$; where c_i denotes the percentage of processing resources allocated to process i. For an allocation to be feasible, $\sum_{i=1\ldots n} c_i \leq 1$; however, it is not necessary to *impose* this constraint explicitly as the actual CPU utilization levels are observed variables, rather than explicitly controlled; hence, the condition is always satisfied. Each of the n processes has an associated bandwidth allocation percentage, denoted by $b_1, b_2, \ldots, b_n$, where b_i denotes the percentage of (local) link bandwidth allocated to process i. Naturally, $\sum_{i=1\ldots n} b_i \leq 1$. In general, CPU utilization of any process, i, $c_i : \Re^n_+ \mapsto \Re_+$, is a function of the bandwidths, $b \in \Re^n$, allocated to *all* processes, and also depends on overall system load, number of concurrent processes and their interactions, memory allocation, choice of network transport protocol, etc.

If the relation of a process' CPU usage to its allocated bandwidth were known, our optimization task would be relatively straightforward; we could then seek to allocate the bandwidth vector, b, that optimizes the goal function, in this case minimizing some norm of the distance between observed CPU percentages and the target. In practice, however, the relation of CPU utilization to bandwidth is not a known and deterministic mapping.

The main contribution of our work is the development of a method for the *joint learning and optimization* of a function of this mapping. We define the problem as one in which the CPU-bandwidth relation is initially some (simple) a priori approximation. Through our adaptive algorithm, the CPU-bandwidth relation is updated iteratively, thereby *learning* the shape of this surface as a function of the control variable, the bandwidth allocation vector, b.

Note that each step in the algorithm's operation will involve assigning a new bandwidth value to each process. Each change of the available bandwidth perturbs the system, and, as such, the number of iterations should be minimized. In addition, the CPU-bandwidth relation is not deterministic; rather it includes many sources of randomness including transient queueing effects associated and variations of a process's CPU utilization due to process state changes as well as random changes in (wide-area) network state.

Figure 1 illustrates the CPU utilization surface of one process as the bandwidth allocations for two processes running jointly on a node are increased. The piecewise-linear form of the curve is due to the sampling granularity in bandwidth space of the data. Notice that the c_2 is increasing in its bandwidth b_2.

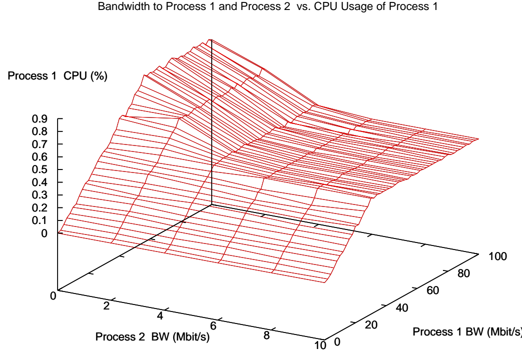

Figure 1: Observed relation between bandwidth allocated to two different processes running concurrently and the CPU usage of one of the processes.

$$f(b) := \frac{1}{2}E\left[\sum_{i=1\dots n}[t_i - c_i(b)]^2\right] = \frac{1}{2}E\|t - c(b)\|^2. \tag{1}$$

The notation E means expected value, and t, b, and c are vectors composed of the t_i, b_i, and c_i, for $i = 1\dots n$, respectively.

We shall optimize a distance criterion such as (1). We refer now to the *stochastic*, learnt function of CPU usage, $C(b)$. The mathematical definition of the learnt CPU usage function will evolve at each iteration, j, of our algorithm. Letting the superscript, j, indicate the iteration, and as before the subscript, i, indicates the process, then for each $i = 1\dots n$, and each iteration j,

$$C_i^j(b_i) = c_i(b_i) + \epsilon_i^j, \tag{2}$$

where we assume that the ϵ_i^j are i.i.d. random variables. The *sequence of functions* $\{C_i^j(b_i)\}$ converges to $c_i(b_i)$ for all $i = 1, \dots n$ as iteration $j \to \infty$. At each iteration, j, we shall thus seek to minimize

$$\begin{aligned}\frac{1}{2}E\sum_{i=1\dots n}(t_i - C_i^j(b_i))^2 &= \frac{1}{2}E\|t - C^j(b)\|^2 \quad (3)\\ &= \frac{1}{2}\|t - \bar{C}^j(b)\|^2, \quad (4)\end{aligned}$$

where $\bar{C}_i^j(b_i)$ is an empirical expectation of the CPU utilization for process i at iteration j. The function (4) must be optimized subject to constraints on the control variable, b. In particular, it must hold that

$$b \in B := \{b \,|\, \sum_{i=1\dots n} b_i \leq 1,\, b_i \geq 0, i = 1\dots n\}. \tag{5}$$

The model defined by (4)–(5) is a stochastic, nonlinear program with polyhedral constraints. As we shall see in the next section, it is globally non-differentiable, due to the construction of the adaptive mapping, but using our approach, the non-differentiability does not hinder the optimization procedure.

4 Adaptive Optimization and Approximation Algorithm

The objective of the algorithm is as follows: given a target CPU allocation vector, starting from an initial bandwidth allocation vector $b_1, b_2, \ldots b_n$, dynamically adjust the bandwidth allocations in such a way that the n processes (applications) consume the target amount of processing capacity.

Processing capacity utilization is indirectly modeled via the functions $C_i(b)$ for every $i = 1 \ldots n$. If these functions are known a priori and deterministic, the optimal $\{b_i\}$ can be found as the solution to a non-linear optimization problem. However, given the large number of parameters and dependencies, as illustrated in the previous section, the exact form of these functions cannot be known in advance. The solution we propose is thus to adaptively construct a learnt model of the $\{C(b)\}$ mapping. Over this evolving set of surfaces, our algorithm searches for the optimal operating point, given by the minimization of the Euclidean distance to the target operating point (4).

4.1 Adaptive model of the CPU utilization mapping, C

As shown in Figure 1, the CPU utilization surface exhibits some regularity, in spite of its complexity. In particular, we observe that $C_i(b)$ increases in b_i. Furthermore, as expected, the CPU utilization of process 1 decreases with increasing bandwidth allocated to process 2. While it is not possible to know the shape of C_i a priori, the algorithm will learn the shape of the mapping.

The learning procedure is based on improving piecewise-linear approximations of each mapping, C_i^j, for each process, i, at each iteration, j. In iteration $j = 1$, the function $C_i^1(b_i)$ has only one slope, α_i^1 and one intercept, γ_i^1. In most cases, the intercept $\gamma_i^1 = 0$. At iteration $j = 2$, the function is composed of precisely two pieces, each with slopes potentially different from α_i^1; the pieces are numbered 2 to 3. Those two pieces have slopes α_i^2 and α_i^3 and intercepts γ_i^2 and γ_i^3. Similarly, the pieces defined at iteration 3 will be numbered 4 to 6, as will their respective slopes and intercepts, and so on.

Specifically, pieces defined at iteration j, C_i^j are given by:

$$C_i^j(b_i) = \tag{6}$$

$$\begin{cases} \alpha^{q(j-1)+1} b_i + \gamma^{q(j-1)+1}, \ 0 \le b_i \le b_i^{q(j-1)+1} \\ \alpha^{q(j-1)+2} b_i + \gamma^{q(j-1)+2}, \ b_i^{q(j-1)+1} \le b_i \le b_i^{q(j-1)+2} \\ \vdots \\ \alpha^{q(j)} b_i + \gamma^{q(j)}, \ b_i^{q(j)} \le b_i \le 1, \end{cases}$$

where $q(j)$ is an iteration-dependent index, defined as follows:

$$q(j) = \sum_{i=1}^{j} i, \tag{7}$$

and the ranges in bandwidth of each piece are provided by the inequalities on each line of (6).

For the system under consideration, the optimization problem (4) exhibits the following properties.

Proposition 1 *The processing power function, $C_i^j(b_i)$ is:*

1. *piecewise linear,*

2. *continuous, and*

3. *nondifferentiable in* b_i.

Proof: The mapping $C^j(b)$ is constructed iteratively. At the first iteration of the algorithm, $C_i^j(b_i)$ is a line passing through the origin $(0, 0)$ and, possibly, the point $(1, 1)$, for each $i = 1, \ldots n$. In each iteration, when a new bandwidth point is determined, the resulting CPU utilization is measured and the slopes on either side of each C_i^j are updated to reflect insertion of the new point. In particular, the mapping, C_i^j will have j linear pieces at iteration j. Consequently, each C_i^j is continuous and piecewise-linear in b_i, its argument. Nondifferentiability follows from the piecewise-linearity.

The CPU utilization function, C_i^j, is modeled as a function of a scalar argument, b_i, only. Although there are clearly cross effects, i.e., C_i indirectly depends on b_j, $j \neq i$, as mentioned previously, the algorithm treats the problem as if the functions were separable across processes, i. The cross effects are explicitly dealt with as stochastic variations, which allows for vastly shorter running times than are required for learning a multi-dimensional model of each mapping, C_i.

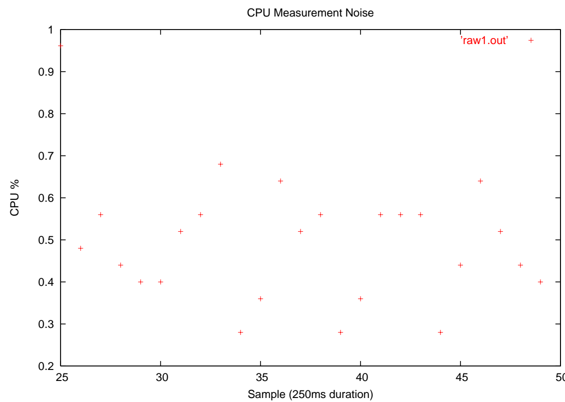

Figure 2: Example of randomness in the CPU usage of one process for a single bandwidth vector. The left side shows variation in CPU utilization shortly after the bandwidth change (after sample 24), and the right side shows variation after hysteresis effects are dissipated.

Randomness in the CPU/bandwidth mapping arises from both time-dependent and time-independent phenomena. An illustration is provided in Figure 2. Observe, for example, from Figure 2 that for a single bandwidth level, b_i, the CPU utilization, C_i varies across readings. The left side of the figure illustrates the hysteresis effect, namely the fluctuations in CPU utilization of a process immediately following the bandwidth change. The right side of the figure shows random fluctuations which occur once the hysteresis effect has dissipated, and reflects in a large part the cross-effects from other processes. That is, the random effects fall into two categories:

- **Transport protocol and time-dependent randomness, or hysteresis**:

  When the bandwidth allocated to a process is increased or decreased, the impact on CPU utilization may take a varying amount of time, depending on the transport protocol, buffer size, etc. For example

if a process is using TCP, addition of bandwidth at the network layer will result in slower addition to the bandwidth perceived by the process, due to TCP additive rate increase. This effect is amplified by the roundtrip delay between the two sides of the process. Time-dependent randomness in the observation of C_i may be introduced by this transport-protocol-related effect.

- **Time-independent stochastic effects**: In addition to time-dependent randomness, there is time-independent noise in measurements due to variation of CPU utilizations by the operating system and by other processes.

We shall make the following assumption for the sake of convergence of the algorithm.

Assumption 2 [Monotonicity of each $C_i^j(b_i)$] *Assume that the mapping $C_i^j : \Re_+^n \mapsto \Re_+$ is increasing in its argument, b_i. for all $b_i \in B$, $i = 1 \ldots n$.*

Note that, typically, a CPU utilization C_i will increase in b_i, but decrease in b_k, for $k = 1, \ldots n$, $k \neq i$. We are not making use, however, of cross derivatives in our algorithm, although cross effects are implicitly present due to the fact that the sum of all CPU utilizations cannot exceed 1. If, for example, $b_2 = 1 - b_1$, we see that if C_1 is increasing in b_1, C_2 is *decreasing* in b_1 by construction.

Remark 1 [Singularity of the Hessian of f] *Due to the presence of the simplex constraint on the bandwidth vector, $\sum_{i=1,\ldots n} b_i = 1$, the Hessian of the objective function (4) is of rank $n-1$, since the nth term of the gradient can be expressed in terms of the other $n-1$ processes. To avoid singularity of the Hessian, it is sufficient to redefine the problem in a reduced space, of $n-1$ processes, the nth process' bandwidth thus being derivable from the remaining $n-1$.*

4.2 Steps of the stochastic Newton-type simultaneous approximation and optimization algorithm

Our approach is to apply the theory of stochastic quasi-gradients and replace the unknown CPU utilization function with a linear, separable approximation of it.

Let $\bar{C}_i(b_i)$ be a smoothed estimate of the CPU utilization for process i, for some number L of sample observations; that is,

$$\bar{C}_i(b_i) = \frac{1}{L} \sum_{l=1}^{L} C_{i,\ell}(b_i, \epsilon_i), \quad (8)$$

We make use of multiple observations of the CPU utilization at a single bandwidth level to obtain an estimate of the expectation of the subgradient of C, ζ, with minimal system perturbation.

See Figure 3 for an illustration of the steps of the algorithm. Conside a single process; hence we will drop the subscript i for the moment. In the initialization step of the algorithm, the initial bandwidth value

b^0 is set, and the smoothed estimate of the CPU level at the bandwidth level b^0 is $\bar{C}^0$. Suppose that $\bar{C}^0 > t$, the upper bound (UB) is set to be the current point, $UB = \bar{C}^0$, and a new bandwidth b^1 is chosen as the intersection of the target CPU level, t, and the estimated function. Next, the CPU usage with bandwidth b^1 allocated is again measured L times, providing an estimate, $\bar{C}^1$. Suppose that it again exceeds the target, t, then the upper bound is reset to the new point, $UB = \bar{C}^1$, and a new bandwidth b^2 is chosen. In the following iteration, suppose that we find that bandwidth allocation b^2 leads to a CPU usage that is below the target $\bar{C}^2 < t$. This new point then becomes the new lower bound, $LB = \bar{C}^2$ and the next bandwidth allocation b^3 is chosen as the intersection of the target CPU line and the line between LB and UB. The process continues until a stopping criterion is satisfied. In the actual algorithm, the movement is moderated by use of a divergent-series step, which serves to slowly dampen the perturbations of the bandwidth.

A detailed description of the algorithm is provided below.

1. Initialization. Let the number of processes be n and the target CPU values be referred to by the $n-$vector, t. Set iteration counter, $j = 1$. Set initial bandwidth vector to a given starting point b_i^0 or set to $b_i^0 = 1/n$ for every $i = 1 \dots n$ if no intial point is provided. Define the initial values $\bar{C}_i^0(LB(i)) = 0$ and $\bar{C}_i^0(UB(i)) = 1$, for all $i = 1 \dots n$. Set stopping criterion threshold $\delta > 0$ to an appropriate value.

2. Main loop. While the stopping criterion has not been reached, repeat:

   (a) Sample the CPU usage of each process i with the current bandwidth vector b. For each process $i = 1 \dots n$, set $\bar{C}_i(b_i)$ to the smoothed CPU usage of process i.

   (b) For each $i = 1 \dots n$, if $\bar{C}_i^j(b_i^j)) > t_i$ then set UB(i)=j. Conversely, if $\bar{C}_i^j(b_i^j)) < t_i$ then set LB(i)=j.

   (c) Direction finding. Determine the search direction, $g^j(b)$ such that $g_i^j(b) = (\bar{C}_i^j(b_i^j) - t_i) * \zeta_i^j = (\bar{C}_i^j(b_i^j) - t_i) * m_i^j$ where m_i^j is the slope of the piecewise-linear function $c_i(b_i)$, evaluated in the direction towards the target value, t_i; i.e., if, for process i, the current iteration counter j is the upper bound, then m_i^j, $i = 1 \dots n$, is the gradient of the segment between point $\bar{C}_i^j(b^j)$ and $\bar{C}_i^{LB(i)}(b^{LB(i)})$. That is $m_i^j = \alpha_i^\ell$ for some active piece ℓ. Conversely, if j= LB(i), then m_i^j is the gradient of the segment between point $\bar{C}_i^j(b^j)$ and $\bar{C}_i^{UB(i)}(b^{UB(i)})$.

   (d) Newton step. The Newton step is given by the (sub)-gradient scaled by the norm of the Hessian. Since we assume our objective to be seaparable, the norm of the Hessian is given, for each process i, by the second derivative of the objective function evaluated at the active piece. Hence, the Newton direction is given by $N_i(b^j) = (1/m_i^j)^2 g_i^j(b_i^j) = (\bar{C}_i^j(b^j) - t_i)/m_i^j$ for all $i = 1 \dots n$.

   (e) Step size. Use a divergent-series step, $s^j = \gamma/(j+1)$, for some scalar constant, γ.

   (f) Update the bandwidth vector: set $b^{j+1} = b^j - g^j(b) * s^j$, and set iteration counter $j = j + 1$.

   (g) Evaluate stopping criterion. For example, if $\sum_{i=1 \dots n} |UB(i) - LB(i)| \leq \delta$, then stop. Else return to step (a) and continue.

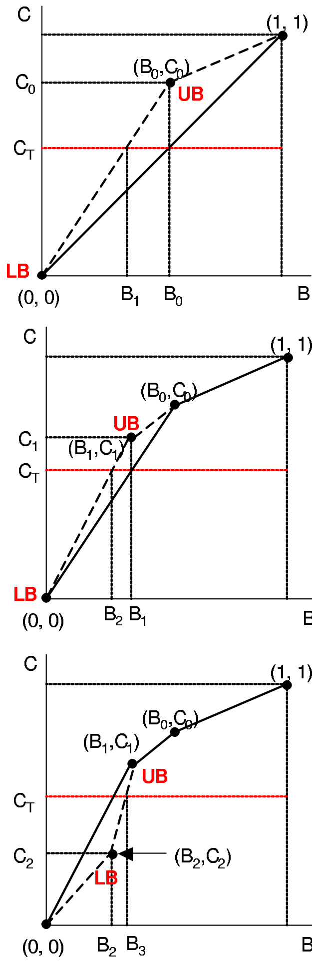

Figure 3: Illustration of the steps of the algorithm.

Remark 2 *A list of some or all $(b_i^j, \bar{C}_i(b_i^j))$ pairs may be kept in a sorted list for each $i = 1 \ldots n$. This makes it possible to quickly look up a bandwidth allocation b_i and estimated local slope m_i when a target t_i changes.*

Proposition 2 *Let*

$$\zeta_i^j(b_i) = E[\xi_i^j], \tag{9}$$

where ξ_i^j is the ith component of a stochastic quasi-gradient of the CPU utilization mapping at iteration j. Then, $\zeta_i^j(b_i) = \bar{C}'(b)$ is a smoothed estimate of the stochastic quasi-gradient of $C^j(b)$ at iteration j.

Proof: Let the current iterate, b_i^j for process i and iteration j be such that $\bar{C}_i^j(b_i^j) > t_i$. Then, an estimate

of the expectation of a subgradient of C_i is, for some number L samples,

$$\zeta_i^j = \frac{1}{L}\sum_{l=1}^{L}\xi_i^{\ell,j}(b_i^j,\epsilon_i), \tag{10}$$

$$= \frac{1}{L}\left[\sum_{l=1}^{L}\frac{C_i^{l,j}-LB^j}{b_i^{l,j}-b^j(LB)}\right], \tag{11}$$

$$= \frac{\bar{C}_i^j - LB^j}{b_i^j - b^j(LB)}, \tag{12}$$

$$= (\bar{C}_i^j)'(b_i^j) \tag{13}$$

where the second line comes from the definition of the subgradient of the piecewise-linear function $\bar{C}_i^j$ on the active segment, LB indicating the lower bound of the active segment.

Under Assumption 2, we have the following property of the algorithm.

Proposition 3 *If for any iteration, j, the search direction vector, $g^j = 0$, then the current iterate, b^j is a solution to optimization problem (4).*

By construction, we have that, for each process $i = 1 \ldots n$, at every iteration, j,

$$\bar{C}_i^j(LB_i^j) \leq t_i \leq \bar{C}_i^j(UB_i^j). \tag{14}$$

In addition, due to the definition of the constraint set B, we have that

$$LB_i \leq b_i \leq UB_i, \tag{15}$$

for every $i = 1 \ldots n$. Under Assumption 2, the slopes of the piecewise CPU utilization mapping, $(C_i^j)'(b^j)^+ \geq 0$ and $(\bar{C}_i^j)'(b^j)^- \geq 0$. Hence, if, at some iteration j, for every $i = 1 \ldots n$,

$$g_i^j = 0, \tag{16}$$

$$(t_i - C_i^j(b^j))m_i^j = 0, \tag{17}$$

then either $t_i - \bar{C}_i^j(b^j) = 0$, $i = 1 \ldots n$, in which case the target has been reached, or $m_i^j = 0$, $i = 1 \ldots n$. In the latter, there are two cases, (i.) $\bar{C}_i^j(b_i^j) \leq t_i \leq \bar{C}_i^j(UB_i^j)$ or (ii.) $\bar{C}_i^j(LB_i^j) \leq t_i \leq \bar{C}_i^j(b_j^i)$. In case (i.) if $m_i^j = 0$, then $\bar{C}_i^j(UB_i^j) = \bar{C}_i^j(b^j)$. Under Assumption 2, $\bar{C}_i^j(UB_i^j) = t_i = \bar{C}_i^j(b^j)$, and analogously in case 2, which completes the proof.

The following assumptions are needed to prove convergence of the algorithm.

Assumption 3 *The sequence of search direction vectors, $\{g^j(b^j,\epsilon)\}$ is bounded. That is, for all j, ϵ, $\|g^j(b^j,\epsilon)\| \leq$*
$upsilon$, for some constant $\upsilon > 0$.

Assumption 4 *[Functional convergence of approximation] The sequence of learnt functions, $f^j(b) \to f(b)$ uniformly over B.*

Assumption 3 holds when the feasible region, B is compact. See Proposition B.24 of Appendix B in [**?**]. Hence, in our problem setting, Assumption 3 is always satisfied. Under Assumptions 3 and 4, we are able to prove the convergence of the algorithm to the set of optimal solutions of the original problem.

Theorem 1 *[Convergence of the algorithm to optimal solution] Suppose that Assumptions 3 and 4 hold. By proposition 1, $\bar{C}_i^j$ are continuous for all $i = 1, \ldots n$. Suppose further that $f(b)$ and the learnt functions, $f^j(b)$, for all iterations j, are convex. Then, $f^j(b^j) \to f(b^*) = \min\{f(b) : b \in B\}$.*

Proof: The feasible set, B, is convex and compact by contruction. The successive bandwidth vectors, b^j are given by the iteration $b^{j+1} = \Pi_B[b^j - s^j g^j(\zeta^j, b^j)]$, where $g^j(\zeta^j, b^j)$ is the expected subgradient of the objective function, $f^j(b^j)$, and the steps s^j satisfy $\sum_{j=1,\ldots\infty} s^j = \infty$, $\sum_{j=1,\ldots\infty} (s^j)^2 < \infty$. Hence, according to [9], the iterations of the simultaneous stochastic optimization and approximation procedure converge to the optimal solution value of the original problem.

The means of making the algorithm robust to the two sources of randomness mentioned in Section 4.1 are different, and are summarized below.

4.2.1 Frequency of observation intervals, transport protocol, and time-dependent randomness, or hysteresis

Iterations take place over (fixed) time intervals based on information obtained over the preceeding interval(s). If the duration of the interval is small, ideally, the algorithm would progress and converge faster. However, as the observation interval shrinks, the quality of the observed CPU allocation measurements deteriorates. Given that there is a lower bound on the amount of time a CPU allocates to a given task, as the interval decreases, the amount of noise increases. Hence, too small an interval can lead to violation of the monotonicity assumption and other unpredictable behavior. On the other hand, too large an interval may lead to very slow convergence.

Furthermore, as previously discussed, the transport protocol and network latency may induce delays in the response of the system. These delays must be taken into account in determining the time duration of each iteration to allow the process to "converge" to the correct CPU utilization.

Hence, to deal with time-dependent random effects, the length of an iteration, in clock time, is treated as a parameter of the algorithm which must also be optimized.

4.2.2 Time-independent stochastic effects and random noise

To counter the effect of noisy measurements due to variation of CPU utilizations by the operating system, each iteration relies on *smoothed* measurements, collected over a number of prior observation intervals. Specifically, we utilize a moving average transformation coupled with a reduction of outlier measurements. Alternatively, an exponential weighted average can be utilized, so that the method is less affected by noise as the interval is reduced.

5 Experimental Design

This section introduces the system we developed to test our adaptive bandwidth control algorithm. Implementing the algorithm in a real-world setting exposed the strengths and weaknesses of the proposed system, and inspired the addition of several features to improve performance.

As in the problem formulation, the system is built around a central controller that uses input from bandwidth and CPU monitoring components to iteratively adjust the bandwidth policing levels until CPU resources are shared in a desired proportion. The system is inherently difficult to control: each iteration is potentially expensive given that it perturbs the operating point, and poorly-chosen bandwidth policing levels can significantly decrease the efficiency and endanger the stability of the computing processes.

In particular, CPU utilization is subject to significant noise levels; this can be due to the granularity of the OS (Linux) kernel scheduler. For example, if CPU utilization is measured over a period shorter than a complete scheduler epoch, the measurement will overstate the CPU share of the processes that have already run while understating the share of those that have not yet run. The next CPU utilization measurement will likely over/understate different sets of processes. This noise can be decreased by measuring CPU utilization over longer measurement periods, somewhat decreasing the responsiveness of the control. Other sources of CPU utilization noise are harder to control, such as the periodic execution of various system services and any other processes not under direct management.

In this test system, as in any real implementation, there is substantial noise in input variables. As previously discussed, CPU utilization measurements over short time periods can be subject to noise from scheduler granularity. In the real system, we also must deal with complications that cannot be modeled as noise. Processes can exhibit different relationships between bandwidth use and CPU utilization over time. This relationship can change quite often and rapidly, as a process' handling of incoming traffic may be entirely dependent on the content of the data. To address this problem, our controller has a mechanism (outside the usual adaptation algorithm) that recognizes a fundamental shift in the bandwidth/CPU relationship, and triggers a "'restart"' of the learning and control process.

5.1 Bandwidth Policing

The bandwidth policing component uses Linux's *tc* command to enforce the bandwidth allocations decided by the controller. The command *tc* (short for "traffic control") is a user-space Linux application that allows a user to configure a set of packet queues, traffic classes, and traffic shapers that reside in the Linux kernel and control the handling of incoming and outgoing packets. *tc* supports a range of packet filters for classifying traffic, queueing disciplines for scheduling packet processing and an efficient mechanism for controlling traffic rates.

5.2 CPU Monitoring

For the results described in this paper, our system management goal is to achieve a given target CPU utilization vector, and our QoS metric is the squared distance of the observed operating point from that vector. CPU utilization is read from Linux's */proc* virtual filesystem, which provides the number of CPU ticks used for each process and by the system as a whole. For any given interval, the ratio of ticks used by a process to the total provides the fractional CPU utilization (regardless of the number of CPUs). This metric is inexpensive to obtain, and accurate for polling frequencies down to tenths of a second [1]

5.3 Bandwidth Monitoring

In addition to creating and deleting filters for policing streams, `tc` allows a user to query a filter for some basic information: bytes delivered, packets delivered, and packets dropped. We use the `bytes delivered` statistic to monitor the amount of traffic associated with each process under management. The measured traffic rate usually differs from the allocated rate, even when an application attempts to use the full amount of bandwidth allocated to it. This difference is attributed to transient effects caused from filter creation/deletion, TCP rate control, and approximations by the rate control filters. These short terms variations make it difficult to use the measured bandwidth in our adaptation algorithm, so we use measured bandwidth only to determine whether or not an application is attempting to utilize all the bandwidth allocated to it. Thus, small changes in the measured bandwidth are not taken to indicate changes in an applications' CPU/bandwidth function; however, when the measured bandwidth is considerably less than the allocated, we use it as an indication that the application is not capable of utilizing the full bandwidth allocation.

[1] Below about 200ms, measurements are inaccurate since the polling frequency approaches the granularity of the Linux scheduler.

6 Experimental Results

In this section we report on results obtained from our experimental set-up. We conducted two sets of experiments; in the first, we evaluated the performance of the method with a set of example stream processing applications in an environment without any other resource allocation schemes. In the second set of results we compared the method against other resource management schemes available today and evaluated how the method could be used in conjunction with these schemes. We start by describing the applications that were used in our experiments.

6.1 Test Applications

In order to produce a realistic testing environment, we experimented with a set of common stream processing applications that exhibit different CPU-bandwidth utilization functions, such as cryptographic, multimedia and text searching applications. In most cases, these are invoked from a scripting language (Python) to facilitate control of the application set.

The first application is stream encryption based on the Blowfish algorithm with a 264-bit key. This application is very CPU intensive; it utilizes approximately 100% CPU to process about 10Mbps of input data. The second application performs text (regular expression) search on an incoming stream, applying filters to find matching regular expressions. As expected, this is a much less CPU intensive application; it can saturate the link bandwidth while using only about 40% of the total CPU. A third multimedia streaming application encodes incoming audio (raw PCM values) into MPEG-II Layer 3 (MP3) format. This application is slightly less CPU-intensive than encryption, and exhausts the CPU at about 16Mbps incoming data rate. The fourth application, a variant on the third, also encodes the same incoming audio to MP3, but at a lower quality setting. This application requires significantly less CPU, and can reach incoming data rates around 55Mbps before exhausting the CPU. Both audio encoding applications have the interesting property of being data-dependent, such that the bandwidth-CPU relation varies somewhat according to the content of the incoming data (e.g. silence is faster to encode than a tonally complex combination of musical instruments).

6.2 Quantitative Results

Figure 4 shows a typical case of the control algorithm at work. One instance of each of the four applications (encryption, search, high quality MP3 encoding, and low quality MP3 encoding) is running on the system, each with a different CPU target. Initially (at iteration zero), each application is allocated up to one fourth of the total bandwidth, which is enough to keep the encryption and mp3 encoding processes busy, constrained by the CPU. In the next iteration (iteration 1), the bottom graph shows the bandwidth allocations to mp3 encode (high quality) and search being decreased, and the upper graph shows the corresponding decrease in CPU usage. Simultaneously, the bandwidth allocation of the encryption process is increased, and the CPU

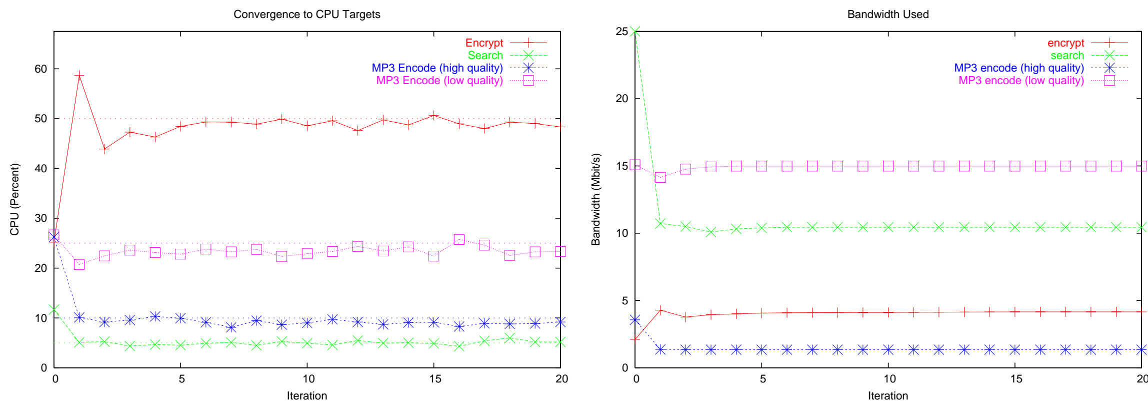

Figure 4: CPU usage (top) and bandwidth usage (bottom) of four processes under control. The CPU targets are: 50% Encrypt, 25% MP3 Encode (low quality), 10% MP3 Encode (high quality), and 5% Search.

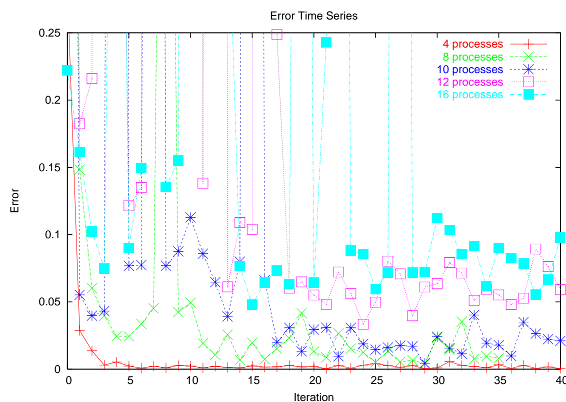

Figure 5: Iteration vs Error (normalized sum of squared distances) for controllers managing 4, 8, 10, 12, and 16 simultaneous processes.

usage decreases. After some vaccilation around the target values, the CPU usage of each process settles in close to the target. CPU utilization for all four applications converges to within 5% of the targets in only three iterations, and stays within that range for the remainder of the experiment.

Figure 6.2 shows the effect of increasing the number of processes under management on the convergence of the processes to their targets. For the cases with more than four processes, there are considerable spikes in error as the system is perturbed toward convergence. Increasing the number of processes under management appears to both increase the time required to reach convergence and to increase the error measured when that convegence is achieved. In this figure, error is measured as the sum of the squared distances of observed CPU usages from their targets, divided by the sum of the squares of the targets. All targets were fixed at 5% CPU, and processes consisted of a mix of the four different applications. Some of the increased error at convergence and the extra time to convergence with added applications is thus attributable to the overall increase in offered CPU load.

6.3 Comparative Experiments

In this subsection we evaluate the performance of the method against other existing alternatives and examine how well it can work in conjunction with these mechanisms. In particular, we compare the method's ability to achieve and maintain a desired CPU allocation with attempts to reach the same allocation using two alternative mechanisms: (i) process priorities in the Linux 2.6 kernel scheduler, and (ii) CPU allocation with the Class-based Kernel Resource Manager (CKRM) patch.

Our goal in the experiments is to maintain a target 3:1 ratio of processing allocation between two instances of the same stream processing application, each instance receiving a separate stream of input data over a shared 100Mbit local area network. We are interested in experimenting with applications that exhibit different bandwidth and CPU resource requirement profiles, from low bandwidth, very CPU intensive to those requiring high bandwidth but relatively low CPU. We therefore created a synthetic stream processing application that can be tuned to model a wide spectrum of applications with different CPU/bandwidth requirements. It achieves this by having a configuration argument that specifies the number of (numerical) computation loops (i.e. *operations*) to execute for each kilobyte-sized block of incoming data. We configure each of the management mechanisms (manually, in the case of `nice` levels, automatically in the case of CKRM and our method) to get as close as possible to the 3:1 ratio target of CPU utilization. We record the observed CPU utilization and incoming bandwidth for at least 3 minutes for each test case, and repeat for each management mechanism. This procedure is repeated for several different settings of the data processing application's argument (number of loops per 1Kb of data). This gives a complete picture of the effectiveness of the different management mechanisms across a wide range of stream-processing applications, varying from high to low bandwidth intensity. To reduce noise in measurements due to background processing, we limited the maximum CPU targets to 75% and 20%, reserving at least 5% of CPU resources for system processes.

6.3.1 Linux Scheduler Priority (`nice`) Testing Procedure

In a stock Linux kernel, the sole mechanism for influencing the relative CPU allocation of running tasks is adjustment of scheduling priorities (also known as the Linux `nice` command, which allows a "nice" user to deprioritize their own tasks and a priveleged user to prioritize tasks). The kernel scheduler uses these priorities to determine both the precedence of and the timeslice given to each task in every scheduling epoch. The `nice` levels range from -20 to 19, with -20 being the highest priority. The kernel adjusts the `nice` level by an interactivity bonus/penalty ranging from -5 to +5, depending on whether a program usually sleeps waiting for some input (bonus) or usually runs (penalty).

Our process for managing CPU allocation using scheduler priorities was to manually search for the pair of priorities for the two running tasks that result in CPU utilizations closest to the targets. For lower values of the processing loops per kilobyte of data, even the most extreme pair of priorities (-20 and 19) were not able to reach the desired CPU targets. In such cases we show the results obtained with these minimum and

maximum priorities.

6.3.2 Class-based Kernel Resource Manager (*CKRM*) Testing Procedure

CKRM is a patch to the Linux kernel that allows “Class-based Kernel Resource Management.” Through a psuedo-filesystem, users can create classes, assign running tasks to them, and set CPU sharing allocations of each class.

Our process for managing CPU allocation using CKRM was to put each task in a separate class, then set the CPU share guarantees (which is also used as a weight for CPU sharing) to the desired targets. As previously mentioned, we allow the sum of the two targets to be at most 95% of the CPU, as leaving less than 5% of the CPU for system tasks and our monitoring components would make the system potentially unstable.

6.3.3 Testing Procedure

Our method can be launched by any user with `sudo` priveleges to run the kernel network QoS controller `tc`. It accepts arguments specifying the names or process identifiers of the tasks to be controlled, and the desired CPU targets.

We configured the system to sample CPU usage every second, and to use 5 of these samples per iteration. In such a configuration, the system may change the bandwidth allocation to a task at most once every five seconds, though in actuality it rapidly converges on an allocation and maintains it unless the system is disturbed.

6.3.4 Comparative Experimental Results

Figure 6.3.4 shows the CPU utilizations achieved by the tested management schemes under a range of the operations-per-kilobyte parameter ranging from 0 to 80. The dotted lines labeled “targets” show the target levels of CPU utilization – the levels we attempted to achieve using each of the tested mechanisms.

First, we note that our method, referenced in the figures as “atm” for “autonomic traffic manager”, is consistently closest to the target levels across the entire spectrum of processing load. The difference is most remarkable when processing load per unit bandwidth is low, i.e., for very bandwidth intensive applications. In the range between 0 and 20 processing loops per kilobyte, neither the Linux Process Priorities approach nor CKRM are capable of achieving any differentiation at all between the two tasks, let alone the desired 3:1 ratio. the Linux Process Priorities approach begins showing the ability to effect

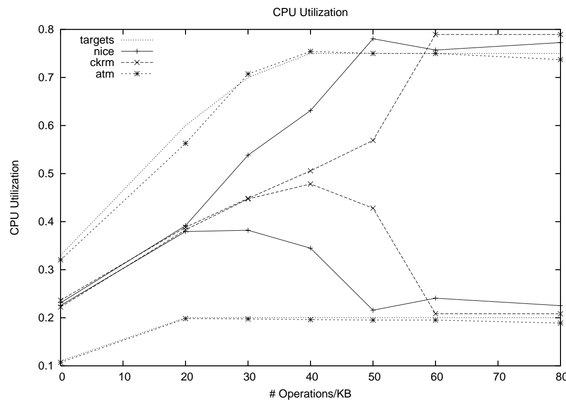

Figure 6: CPU Utilization measured using each of the 3 management schemes: nice, ckrm, and our approach, called "atm", for varying processing levels. Each scheme is represented by two lines for the two tasks, and the target CPU utilizations are indicated by the dotted lines labeled "targets."

differentiation above 20 loops/KB, and reaches the target differentiation at about 50 loops/KB. CKRM shows very little differentiation below 40 loops/KB, and reaches the targets at about 60 loops/KB.

We believe that the Linux Process Priorities approach and CKRM are unable to achieve differentiation in the case of low CPU/high bandwidth intensity applications because the processing tasks receive data via TCP/IP over a shared link, and the TCP/IP stack does not explicitly favor tasks of higher `nice` priority or larger CKRM CPU allocation. Thus TCP/IP tends to equalize the bandwidth allocated to each of the two tasks unless another mechanism is used to explicitly control this allocation, which is exactly what our method does.

For cases with 60 loops/KB and above, all three mechanisms are able to maintain the system at target levels (note that the Linux Process Priorities approach and CKRM converge on CPU utilizations slightly above the targets, since they will redistribute any unused CPU time above the 95% we allocate). As the loop count increases, tasks become increasingly CPU-dependent, and pure CPU-allocation scheduling schemes can successfully maintain the desired balance, while management by pure bandwidth-allocation schemes such as our method become more difficult. Nevertheless, we find that our method is quite capable of reaching CPU targets in separate trials with up to 10,000 processing loops/KB.

Figure 6.3.4 shows the bandwidth utilization measured during the tests. Similar to the CPU utilization graphs, we see that the Linux Process Priorities approach and CKRM do not achieve any meaningful differentiation between the two tasks for loops/KB values below 20. Bandwidth differentiation closely parallels CPU differentiation: the Linux Process Priorities approach shows some differentiation above 20 and converges with our method (referred to as "atm") at 50, while CKRM shows differentiation above 30 and converges with our method and the Linux Process Priorities approach at 60.

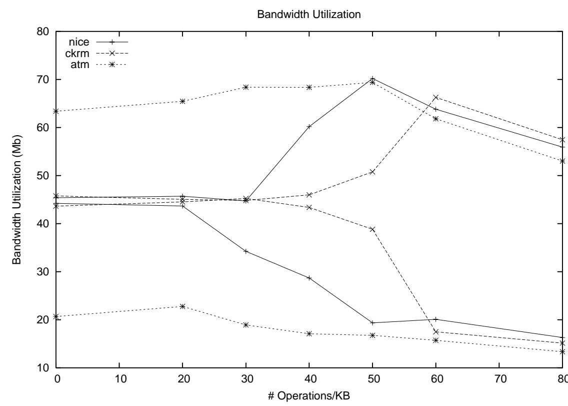

Figure 7: Bandwidth Utilization measured using each of the 3 management schemes: nice, ckrm, and atm for varying processing levels. Each scheme is represented by two lines corresponding to the two tasks.

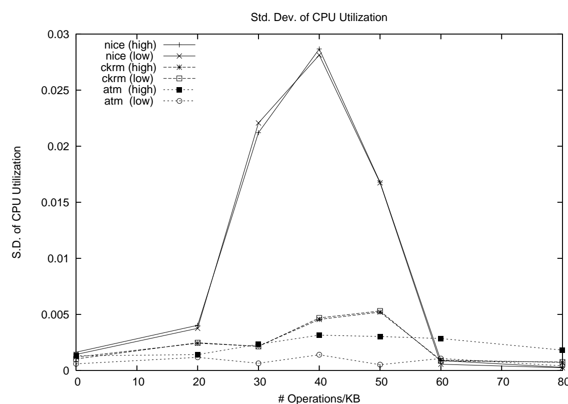

Figure 8: Standard deviation of CPU utilization.

The bandwidth utilization lines for our method are essentially flat for the section of the line between 0 and 40 loops/KB, reflecting the fact that targets increase in direct proportion to the number of instructions per KB. Above 50 loops/KB, the CPU is fully utilized, so increasing the loops/KB variable leads to linear decrease in the bandwidth. The bandwidth utilizations are very near the ideal, as can be observed from the fact that the Linux Process Priorities approach and CKRM converge to our method's bandwidth utilization once they reach the targets.

Figure 6.3.4 shows the standard deviation of all samples taken during all trials of `nice`, CKRM, and our method. The predominant feature of the graph is a large peak demonstrated by `nice` in the range between 20 and 60 operations per kilobyte. Recall that this range corresponds exactly to the region where the Linux Process Priorities approach transitions from achieving no differentiation between tasks to achieving the full targeted differentiation. In fact, during experimentation we noticed that the Linux Process Priorities

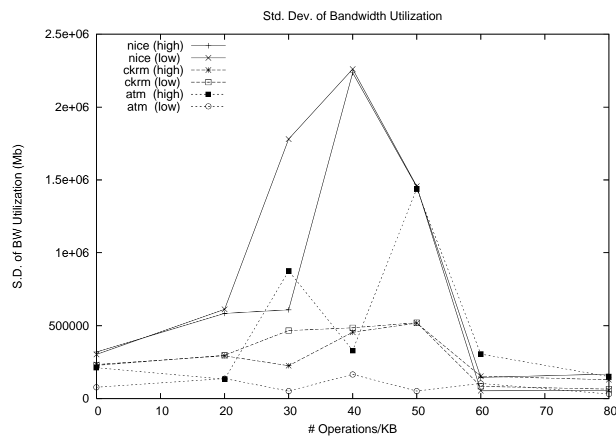

Figure 9: Standard deviation of bandwidth utilization.

approach was producing very unstable system operation in this range around 40 operations per kilobyte, with both CPU utilization and bandwidth usage oscillating quite violently. This may be the result of interaction between the Linux scheduler giving priority and larger timeslices to a task while TCP/IP detects higher losses on that tasks' link, and repeatedly throttles it down (via its multiplicative decrease algorithm).

CKRM exhibits standard deviation which increases roughly linearly from 0 to 50 operations per kilobyte, then drops near zero at 60 operations per kilobyte and thereafter. Our method, in comparison, shows standard deviation roughly flat across all operations/KB, demonstrating that it is not very sensitive to the ratio of bandwidth to CPU. For 0 to 50 operations per kilobyte, ATM exhibits similar or less variance around the CPU targets than either the Linux Process Priorities approach or CKRM. Both CKRM and the Linux Process Priorities approach perform very well when tasks are sufficiently CPU-centric: with 60 or more operations per kilobyte, the Linux Process Priorities approach and CKRM exhibit less variance than our method.

Figure 6.3.4 shows the standard deviation of the bandwidth used across all samples for each trial. Similar to Figure 6.3.4, we observe that `nice` exhibits relatively large variance in the region between 20 and 60 operations per kilobyte. Our method displays differing variance in different trials, at times quite high for the higher priority task's bandwidth, but consistently lowest of the group for the lower priority task's bandwidth. We attribute the former observation to our method's bandwidth search mechanism, which likely had to adjust bandwidth allocation rapidly to maintain the desired CPU levels in this challenging operating region. In observing our method at work, we often see that system tasks can "'steal"' CPU from the stream processing tasks (since our method, unlike the Linux Process Priorities approach and CKRM, does not directly control CPU scheduling). This is consistent with the observation of low variance in the low priority task under our method, since it is less likely to have CPU share stolen by system tasks, and can lock in a specific bandwidth and maintain that allocation for long periods of time.

We attempted to run both our method and CKRM simultaneously, with no coorindation between the two, and found that the combination did not perform as well as our method alone. Specifically, our method seems to experience higher variability and slightly less differentiation in the presence of CKRM than in isolation. With our method's ability to control highly bandwidth-centric tasks and CKRM's ability to control CPU-oriented tasks, using the two in concert, in a coordinated fashion, or dynamically choosing between them, should enable improved control across the full spectrum of stream processing tasks, and with other types of processes as well. Determining the best way to do so would present a worthwhile avenue for future study.

7 Conclusions

Efficient operation of distributed and parallel computing applications depends on efficient management of multiple resources such as CPU, system memory and link bandwidth. While substantial work exists on *independently* managing system (CPU, memory) and networking resources, little attention has been given to the complex dependencies between utilization of processing and network bandwidth resources. As our work shows this link is a critical piece of system management tools, especially in the case of high-bandwidth stream processing applications, for which existing system management tools based on CPU-prioritization schemes are inadequate.

To address these shortcomings, we have developed a system that seeks to achieve system management objectives by explicitly considering the relationship between the bandwidth allocated to an application and its corresponding utilization of processing resources. By controlling the bandwidth allocation vector across different applications, we can drive the system towards a desired CPU utilization vector. Effective operation of our system has been demonstrated for a wide-range of applications, from CPU intensive to bandwidth-intensive.

Our results demonstrate that weighted and prioritized CPU scheduling are not sufficient to achieve meaningful control over high-rate stream processing operations, and that our method can achieve such control over both high-rate (low operations/unit of bandwidth) and low-rate (high operations/unit bandwidth) stream processing tasks. Our method controls CPU utilization indirectly by setting the bandwidth to (or from) each processing tasks, and so must learn the relationship between the bandwidth and CPU resources used by each task, while simultaneously effecting the desired control output. Despite the algorithmic sophistication of our method, when compared to simple weighted CPU sharing (e.g. Linux Process Priorities (setting juudiciously the `nice` command) and CKRM), our method achieves accuracy consistency comparable to those two approaches for low-rate stream processing applications, and, most importantly, improves upon their performance with the high-rate stream processing tasks that the Linux Process Priorities approach and CKRM cannot handle. Furthermore, our method achieves this will notably low overhead.

We anticipate that there will be gains made by managing multiple resources in a single framework, above and beyond those that we achieved here by controlling different resources via entirely isolated mechanisms. Hence our suggestions for future research in the are are two-fold. On the one hand, a study of incorporating

our bandwidth-based traffic management approach in conjunction with other existing traffic management schemes (such as Linux Process Priorities, CKRM, or others) would present an important contribution to the state of the art. On the other hand, our ideas could be developed further by designing schemes to handle more than one resource. For example, network bandwidth and CPU shares, and/or memory could be jointly controlled for still greater benefit to stream processing systems and other I/O centric applications.

References

[1] L. Abeni, G. C. Buttazzo, "Adaptive Bandwidth Reservation for Multimedia Computing", In Proceedings of 6th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA), 1999.

[2] D. Bertsekas, *Nonlinear Programming*, Athena Scientific, Belmont, Massachusetts, 1999.

[3] R. Braden, D. Clark, S. Shenker, "Integrated Services in the Internet Architecture: an Overview", RFC1633, June 1994.

[4] S. Chandra, C. S. Ellis, and A. Vahdat, "Application-Level differentiated multimedia web services using quality aware transcoding", IEEE Special Issue on QOS in the Internet, 2000.

[5] "Class-based Kernel Resource Management (CKRM)", http://ckrm.sourceforge.net/.

[6] E. Crawley, Ed., L. Berger, S. Berson, F. Baker, M. Borden, J. Krawczyk, "A Framework for Integrated Services and RSVP over ATM", RFC2382, August 1998.

[7] R. Cruz, "A Calculus for Network Delay, Part II: Network Analysis," IEEE Transactions on Information Theory, pp. 132–141, 1991.

[8] T. Deane, G. Haff, J. Enuice, "VMware on the March", Research Note, Illuminata, Inc., available at http://www.vmware.com/pdf/illuminata.pdf.

[9] Ermoliev, Y. "Stochastic quasi-gradient methods", Chapter 6 of Numerical Techiniques for Stochastic Optimization, Y. Ermoliev and R. J-B. Wets, Eds., Springer-Verlag, Berlin, 1988.

[10] A. Fox, "Adapting to Network and Client Variability via On-Demand Dynamic Distillation", in Proceedings of Seventh Intl. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS-VII), Cambridge, MA, 1996.

[11] L. Georgiadis, R. Gurin, V. Peris, K.N. Sivarajan, "Efficient Network QoS Provisioning Based on per Node Traffic Shaping", IEEE/ACM Transactions on Networking, 1996.

[12] A. Goel, M. H. Shor, J. Walpole, D. C. Steere, C. Pu, "Using Feedback Control for a Network and CPU Resource Management Application", In Proceedings of the 2001 American Control Conference, Alexandria, Virginia, June 2001.

[13] R. Guerin, H. Ahmadi, and M. Naghshineh, ”Equivalent Capacity and Its Application to Bandwidth Allocation in High-Speed Networks”, IEEE Journal on Selected Areas in Communications, vol. 9, no. 7, pp 968-981, 1991.

[14] J. Guo, L. N. Bhuyan, ”Load Sharing in a Transcoding Cluster”, In Proceedings of IWDC, pp. 330-339, 2003.

[15] E. Knightly and H. Zhang, “D-BIND:an accurate traffic model for providing QoS guarantees to VBR traffic”, IEEE/ACM Transactions on Networking, Volume 5, Issue 2, pp 219 - 231, April 1997.

[16] J. Liang, K. Nahrstedt and Y. Zhou, ”Adaptive Multi-Resource Prediction in Distributed Resource Sharing Environment”, In Proc. of fourth IEEE/ACM Symposium on Cluster Computing and the Grid (CCGrid’04), Chicago, IL, April, 2004.

[17] ”LPAR: Dynamic Logical Partitioning, An IBM Virtualization Engine Systems Technology”, http://www-1.ibm.com/servers/eserver/iseries/lpar/.

[18] K. Nichols, V. Jacobson, L. Zhang, ”A Two-bit Differentiated Services Architecture for the Internet”, RFC2638, July 1999.

[19] D. Rosu, K. Schwan, S. Yalamanchili, R. Jha, ”On adaptive resource allocation for complex real-time applications”, In Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS ’97), December 1997.

[20] D. Steere, M. H. Shor, A. Goel, J. Walpole, C. Pu, ”Control and modeling issues in computer operating systems: resource management for real-rate computer applications”, In Proceedings of 39th IEEE Conference on Decision and Control (CDC2000), Sydney, Australia, December 2000.

[21] L. Xiao, M. Johansson, H. Hindi, S. Boyd, and A. Goldsmith, ”‘Joint optimization of communication rates and linear systems”’, In *Proceeedings of 2001 Conference on Decision and Control*, December 2001, session WP07-4.