*p*Clock: An Arrival Curve Based Approach For QoS Guarantees In Shared Storage Systems

Ajay Gulati
Rice University
6100 Main St
Houston, TX 77005
gulati@rice.edu

Arif Merchant
HP Labs
1501 Page Mill Rd
Palo Alto, CA 94304
arif@hpl.hp.com

Peter J. Varman
Rice University
6100 Main St
Houston, TX 77005
pjv@rice.edu

ABSTRACT

Storage consolidation is becoming an attractive paradigm for data organization because of the economies of sharing and the ease of centralized management. However, sharing of resources is viable only if applications can be isolated from each other. This work targets the problem of providing performance guarantees to an application irrespective of the behavior of other workloads. Application requirements are represented in terms of the average throughput, latency and maximum burst size. Most earlier schemes only do weighted bandwidth allocation; schemes that provide control of latency either cannot handle bursts or penalize applications for their own prior behavior, such as using spare capacity.

Our algorithm *p*Clock is based on arrival curves that intuitively capture the bandwidth and burst requirements of applications. We show analytically that an application following its arrival curve never misses its deadline. We have implemented *p*Clock both in DiskSim [2] and as a module in the Linux kernel 2.6. Our evaluation shows three important features of *p*Clock: (1) benefits over existing algorithms; (2) efficient performance isolation and burst handling; and (3) the ability to allocate spare capacity to either speed up some applications or to a background utility, such as backup. *p*Clock can be efficiently implemented in a system without much overhead.

Categories and Subject Descriptors: C.4 [Performance of Systems]: Modeling techniques; D.4.2 [Operating Systems] [Storage Management] : Secondary storage; D.4.8[Operating Systems] [Performance] : Modeling and prediction

General Terms: Algorithms, Design, Management, Performance

Keywords: Burst handling, Fair scheduling, QoS, Real time guarantees, Resource allocation, Storage performance virtualization

1. INTRODUCTION

Consolidation of storage belonging to one or more organizations in a centralized shared repository is an increasingly popular paradigm for managing organizational or departmental-level data. The advantages of this approach include the ease of centralized management, flexibility in data placement, and lower operating costs. Companies (like Amazon S3 [1] for instance) are beginning to provide storage as a service, where a customer can buy a specific amount (GBytes) of storage with certain reliability and access requirements. Even within an organization, the need for data sharing among different organizational units favors the use of centralized data repositories over ad hoc partitioning and replication of the data sets. However, sharing resources is only practical if the system can provide suitable isolation among the clients in terms of security, privacy and performance.

In this work we focus on obtaining performance isolation among concurrent workloads sharing the resources of a storage server. Each workload must be provided the abstraction of having its own dedicated server with a guaranteed minimum level of performance. Conventionally, sharing of storage resources between workloads with different requirements is handled by statically partitioning the resources, for example, by ensuring that the data for different workloads resides on different disks within a disk array. However, static partitioning is inflexible in its ability to adjust to workload variations, leading to over provisioned and unnecessarily expensive designs. An alternative approach is to share the storage resources between workloads, using statistical multiplexing to reduce the overall resource requirements, and apply scheduling methods to logically isolate the workloads in a way that avoids destructive interference between them.

The performance requirements of a workload are usually expressed in terms of *throughput* and/or *latency* constraints. For example, a multimedia workload might have strict latency requirements in order to provide glitch-free service, whereas a bulk file transfer might care more about overall throughput than the latency of individual requests. Similarly an online transaction and query processing system may need to guarantee fast response time for transactions while providing minimum throughput levels for queries.

The desired capabilities of a scheduling framework are: (1) Meet throughput requirements and deadline guarantees for every workload that follows a stipulated service level agreement (2) Handle bursts of stipulated maximum size without compromising deadline guarantees (3) Flexibly allocate unused system capacity to workloads that desire more than their stipulated service without penalizing their contractual guarantees (4) Allocate spare capacity to a background activity (such as a backup utility or defragmenter) in large bursts, without hurting contractual flows (5) Be work conserving, *i.e.* the system should not idle if requests are pending.

In this paper we present a scheduling scheme ***p*Clock**, that meets the above requirements. The scheduler enables all well-behaved clients (those satisfying their burst and throughput constraints) to meet their stipulated deadlines. We prove that under certain precisely characterized assumptions of deterministic behavior, no well behaved flow will miss its deadlines. In practice the assumptions

SIGMETRICS'07, June 12–16, 2007, San Diego, California, USA.

may be occasionally violated due to the stochastic nature of low-level request scheduling in a storage system, resulting in some missed deadlines. However, as we show empirically, the requirements are met on the average. Our scheduling scheme *p*Clock does not require exact knowledge of the system capacity, which is notoriously difficult to estimate and varies dynamically in a storage server. An estimate of the system capacity is necessary for higher-level admission control. To guarantee hard deadlines, conservative estimates of the capacity and behavior of the flows need to be made. In practice, the system may use less conservative "typical" values and employ a statistical model of flow behavior, providing probabilistic rather than hard guarantees. The modeling and analysis of statistical QoS is not considered in this paper.

A second contribution of the paper is in designing a flexible method to allocate spare capacity of the storage system to background activities. Our scheme allows us to identify chunks of unused capacity that can be allocated to background jobs, while guaranteeing that future requests of well-behaved flows will still be serviced within their guaranteed bounds. In contrast, previous schemes provide service to background jobs only in small granularities whenever there are no foreground jobs to be run. Alternatively, the spare capacity may be used to accelerate contractual clients. However, this needs to be done in a way that ensures these clients are not penalized later for their use of this spare capacity.

The rest of the paper is organized as follows. We present related work and discuss the limitations of existing techniques in the next Section. In Section 3 we describe the system model and related definitions and then we present our scheduling algorithm, *p*Clock. Formal results are summarized in Section 4. Section 5 presents evaluation results using both *DiskSim* [2] and an implementation of *p*Clock as a Linux kernel module. We end with some conclusions and future directions in section 6.

2. OVERVIEW

We model the storage system as a server with internal concurrency, providing shared service to a set of m clients or *flows*, $f_1, f_2, \cdots, f_m$ and to a background utility, as shown in Figure 1. Requests from a flow wait in a private queue before being sent to the server. We treat the storage server as a black box; it may have an internal queue that reorders the requests that have been dispatched to it. Hence, our scheduler provides strict isolation in terms of queuing delays observed by the competing flows before their requests are dispatched to the server. Once dispatched, the request will finish within a small amount of time assuming an efficient, starvation avoiding scheduler (such as C-LOOK, C-SCAN etc.) at the storage server.

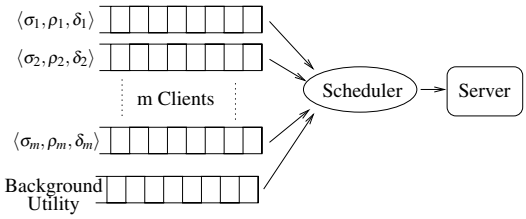


Figure 1: System Model

The service requirements of f_i are specified by a 3-tuple $(\sigma_i, \rho_i, \delta_i)$, where σ_i is the maximum burst size (number of IOs), ρ_i is the throughput (in IOs per second – IOPS) and δ_i is an upper bound on the latency of an IO request (in ms). To guarantee the latency constraints of a flow it is necessary to bound not just the maximum burst size but also how frequently bursts can occur. Otherwise a flow that makes continuous bursts of maximum size could quickly exceed the server capacity, leading to unbounded latencies. The *arrival function* for f_i denoted by $R_i(s,t)$ is the total number of IO requests made by f_i in the time interval $[s,t]$. We say that f_i is *well behaved* if $R_i(s,t) \leq \sigma_i + \rho_i(t-s)$, for all time intervals $[s,t]$. This model of arrivals is also known as a leaky bucket model [20, 23] with parameters (σ_i, ρ_i). The leaky bucket model limits the burstiness of a flow by controlling the size and the frequency of the bursts: the maximum burst size is σ_i and the average arrival rate is ρ_i. The following condition states the requirement for isolation among the flows: *a well behaved flow should never miss its deadline regardless of the behavior of other flows*. Additionally, we would like to meet other goals enumerated in the last section such as the flexible use of spare capacity to speed up contractual flows or background jobs, and work conservation. Note that we do not assume that flows are voluntarily well behaved, nor do we throttle them to conform to the leaky bucket parameters. Forcing the flows to be well behaved allows for simpler scheduling algorithms, but can result in low system utilization. Instead, flows are allowed to opportunistically exceed their arrival function specifications. While the system cannot guarantee meeting the deadlines of the flows with extra requests, in practice it will often be able to use the unused capacity arising from statistical variations in the arrivals to meet the requirements. For instance, in a closed-loop system, where a client can adapt its input rate based on the response time to gain additional service during periods of light load. Since the scheduler isolates the flows, allocating the spare capacity does not interfere with the remaining flows; clients that stay within their arrival specifications will continue to receive guaranteed service.

Most bandwidth allocation algorithms that are used to provide throughput guarantees are based on assigning *tags* to arriving requests. The tags are based on the priority of the flow, arrival time of the request, and its service demand. The queued requests are then serviced in order of their tags. Our algorithm *p*Clock is also based on the use of tags, but has several advantages over the earlier schemes. We now describe three idealized cases to illustrate the benefits of the *p*Clock algorithm. The examples describe simple situations to make them easier to follow; nonetheless, the underlying advantages remain even under more complex system behaviors. These cases respectively illustrate goals 2, 3 and 4 described in Section 1.

Case 1 – Ability to handle bursts: Consider two flows f_A and f_B that have stipulated throughputs of $\rho_A = \rho_B = 50$ IOPS and worst-case latencies $\delta_A = 500$ms, $\delta_B = 250$ms respectively. The maximum burst sizes of f_A and f_B are $\sigma_A = 0$ and $\sigma_B = 25$ IOs. Assume the system capacity is 100 IOPS. Suppose that flow f_A sends requests at a uniform rate of 50 IOPS, whereas f_B sends a burst of 25 IOs every 0.5 seconds. This is shown as Case 1 in Figure 2.

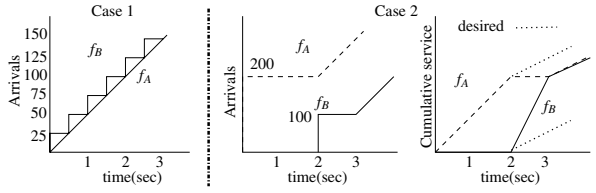


Figure 2: Case 1: f_B is bursty and therefore misses its deadline (250 ms), whereas f_A meets its deadlines. Case 2: f_A is starved during [2,3] for using spare capacity during [1,2]

Bandwidth allocation algorithms (e.g. WFQ [8] and its successors [3,9,10]) will successfully provide both f_A and f_B their desired throughput of 50 IOPS. However they will not be able to control the latencies of the requests. The tags assigned by WFQ [8] (for instance) to requests of both f_A will and f_B will be $1/50$, $2/50$, $3/50$, $\cdots$. Therefore WFQ will interleave the requests of f_A and f_B in essentially a $1:1$ ratio. Requests from f_A will be serviced with very little queuing delay. However, the requests of f_B in each burst will face a worst-case latency of 500 ms. In contrast, *p*Clock will meet the deadlines of both the flows, and all requests from f_B will finish within 250 ms of their arrival, by scheduling requests in a different order.

Case 2 – Penalizing flows for use of spare capacity: Consider two flows f_A and f_B with stipulated throughput $\rho_A = \rho_B = 50$ IOPS. Assume the system capacity is 100 IOPS and burst sizes $\sigma_1 = \sigma_2 = 0$. Suppose that f_A sends a burst of 200 IOs at $t = 0$, and then sends requests at its stipulated rate of 50 IOPS from $t = 2$ onwards, while f_B sends a burst of 100 IOs at $t = 2$, and then sends requests at a steady rate of 50 IOPS after $t = 3$. During the time interval $[0,2]$ f_B is idle and the system has a spare capacity of 50 IOPS. The input profiles are shown as Case 2 in Figure 2.

In the interval $[0,2]$ the spare system capacity is absorbed by f_A, which receives 100 IOPS, well above its stipulated rate of 50 IOPS. When f_B becomes active at time 2, existing scheduling algorithms like Virtual Clock [29] and SCED [23] (see Section 2.1 for further details) penalize f_A for the excess service it received during the first two seconds. Intuitively this is because the tags of f_A will become very high due to the extra requests that were serviced in the first two seconds, and f_A is subsequently starved while the tags of f_B catch up. During the interval $[2,3]$, these algorithms will give all the service to f_B and starve f_A, even though it is f_B and not f_A that is exceeding its service agreement at this time. In contrast, *p*Clock does not penalize a flow that has gained extra service by using spare capacity. For this example, the service provided by *p*Clock after $t = 2$ is shown by the dotted lines in Case 2 of Figure 2; as can be seen both flows receive 50 IOPS.

Case 3 – Flexible allocation of spare capacity: Spare capacity can be allocated in two ways: one is to give all the spare capacity to contractual flows and the second is to use spare capacity to speed up background tasks. Our algorithm allows both these policies. Furthermore, we are able to allocate spare capacity to background tasks in batches rather than in small chunks. Because many background activities (such as backup) perform mainly sequential IOs, system throughput usually increases if the scheduler can provide service to the background job in larger bursts instead of fine-grained interleaving with the contractual flows.

2.1 Related Work

The work related to QoS-based resource allocation can be divided into three broad categories. First is a class of scheduling algorithms for proportionate bandwidth allocation, such as PGPS [20], Virtual Clock [29], WFQ [8], WF^2Q [3], SFQ [10], SCFQ [9], Leap Forward [27] and Latency-rate scheduling [25]. As discussed below, the primary goal of these algorithms is to divide up the bandwidth in a specified manner between the flows; they do not explicitly attempt to control the latencies of the requests. The second class of scheduling algorithms based on survice curves [7, 19, 23, 26] attempt to simultaneously control both the bandwidth allocation and the latencies of the flows. Finally, storage-specific methods such as Façade [17], Stonehenge [13], SFQ(D) [15] and Avatar [28], either use variants of virtual-time based tagging or a feedback based mechanism to provide heuristic assurances. Table 1 provides a quick comparison of *p*Clock with existing algorithms in the three categories.

2.2 Fair Queuing Algorithms

Bandwidth allocation algorithms assign tags to requests based either on *real time* or *virtual time*. An early solution, Virtual Clock, assigned tags to requests based on *real time* [29]. However, it is known that this solution can result in starvation. Other proportional bandwidth schedulers, WFQ [8], WF^2Q [3], SFQ [10, 11], and SCFQ [9], use the notion of *virtual time* tagging to simulate the idealized fine-grained multiplexing of the Generalized Processor Sharing (GPS) algorithm [12, 20]. While these algorithms provide very good bandwidth allocation, the worst-case latency experienced by a request can be high since it depends on the number of active flows [4]. Bennett and Zhang [4], Leap Forward [27], and Latency rate scheduler [25] significantly improved the latency bounds, so that request delay depended only on a flow's own bandwidth allocation and request rate.

A fundamental limitation of these schemes is that it is *not possible to independently control* the bandwidth and latencies of the flows. The latency incurred by a request is inversely related to its bandwidth allocation, so that flows with high (low) bandwidth allocation receive low (respectively high) latency. These algorithms have just one parameter to adjust both throughput and latency, and this is insufficient [7] for meeting both constraints independently.

Other issues that are not handled well by these schemes include the ability to handle bursts and the difficulty of relating virtual time tags to predict free system capacity.

2.3 Algorithms based on Service Curves

To decouple bandwidth allocation and latency requirements, Cruz et al. [7, 23] extended the service curves concept introduced by Parekh and Gallagher [20, 21] to allow variable rates. Using these service curves both bandwidth and latency constraints may be satisfied provided certain capacity constraints are met. They provided SCED [7, 23] algorithm to schedule workloads specified by a given set of service curves that meet the capacity constraints. However, a drawback of their solution is that a client that uses spare system capacity may get starved in the future when resource contention is high. The solution has both the drawbacks described in Case 2 and Case 3 in Section 2. In addition, SCED is unduly conservative in setting the deadlines of requests. It sets the deadline to be the earliest value possible without causing other flows to miss their deadlines; by contrast, *p*Clock sets the deadline of a request to be as late as possible. This permits greater flexibility in scheduling spare capacity (Case 3 of Section 2).

Stoica et al [19, 26] have identified cases where SCED may fail and have provided a modified virtual-time based algorithm to avoid starving a client for using excess capacity. However they don't guarantee that the client can meet its deadline if it uses the excess capacity. Furthermore their scheme may also sometimes cause well-behaved clients to miss their deadline.

2.4 Storage Specific Methods

For storage systems, Façade [17], Stonehenge [13], SFQ(D) [15], and Avatar [28] propose virtual-time based scheduling strategies, while incorporating issues specific to storage workloads. Most of these schemes use virtual tags and a single set of weights to control the throuhgput and the response time. However, it is not possible to control both bandwidth and latency independently with a single parameter, and the use of virtual time makes estimation of spare capacity and real-time deadlines difficult. SCAN-EDF [22] proposes hybrid low-level disk scheduling algorithms to meet request deadlines while maintaining high throughput. Cello [24] and YFQ [5] also provide fair bandwidth allocation mechanisms along with optimizing disk scheduling for seek minimization. SLEDS [6] uses

Algorithm class	B/W allocation	Latency control	Burst handling	Avoid starvation	Spare capacity control	Tagging mechanism
Fair Queuing Algorithms	Yes	No	No	Yes	No	Virtual Time
Service Curve based	Yes	Yes	Yes	No	No	Real Time
Storage specific heuristics	Yes	No	No	Yes	No	Virtual Time
*p*Clock	Yes	Yes	Yes	Yes	Yes	Real Time

Table 1: Comparison of existing scheduling techniques

live feedback from the system to throttle flows based on a leaky-bucket model. Throttling IOs may lead to non work conserving schedules. It is also difficult to give fine grained guarantees on latencies or to handle bursts in SLEDS. Avatar [28] is a two-level scheme that uses EDF scheduling at the second level to meet response time deadlines; however it does not provide guarantees. A control-theoretic approach for scheduling in storage systems was proposed in [16].

3. SCHEDULING FRAMEWORK

In this section we will present our scheduling algorithm and show how each of the goals is met using our framework. We will first introduce terminology and definitions followed by an intuitive description of the components of the algorithm. We then formally present our scheduling algorithm *p*Clock and show that it meets the desired goals. Finally, we discuss the estimation and allocation of spare system capacity.

3.1 Definitions

To simplify the presentation, time is represented by discrete time steps $t = 0, 1, 2, \cdots$. The requirements of a flow f_i are represented by a triple $(\sigma_i, \rho_i, \delta_i)$ where σ_i is the largest number of IO requests that f_i can make at any time step, ρ_i is the average throughput (IOPS) and δ_i (ms) is the maximum allowable latency of a request. The number of requests made by f_i at time step t is denoted by $r_i(t)$. The *arrival function* $R_i(a,b) = \sum_{t=a}^{b} r_i(t)$, is the total number of IO requests made by f_i in the interval $[a,b]$. The amount of service (number of IOs) provided to f_i in the interval $[s,t]$ is denoted by $S_i(s,t)$. In order to meet the performance guarantees the system must service at least $R_i(s,t)$ IOs of f_i within the interval $[s, t+\delta_i]$; *i.e.* $S_i(s, t+\delta_i) \geq R_i(s,t)$.

Flow f_i is well behaved if the size and frequency of its bursts are limited by a leaky bucket with parameters (σ_i, ρ_i). That is, for every time interval $[s,t]$, $R_i(s,t) \leq \sigma_i + \rho_i(t-s)$. This constraint is difficult to use computationally in this form: to determine how many requests f_i can make at time t while remaining well behaved, the definition requires us to check the inequality for all possible intervals ending at t. Hence, the constraint is expressed in the form of a function called the *Arrival Upper Bound* (AUB) that captures the size of the burst permitted at time t succinctly, as defined below.

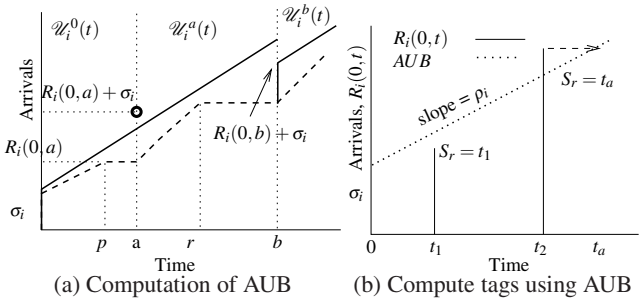
DEFINITION 1. *The* **backlog** *of a flow f_i at time t is denoted by $B_i(t) = R_i(0,t) - S_i(0,t)$. Flow f_i is said to be* **busy** *at time t if $B_i(t) > 0$. A flow that is not busy at time t is* **idle**. *A* **system busy period** *is a maximal-sized time interval during which at least one flow is busy.*

DEFINITION 2. *Flow f_i is* **active** *at time t if $R_i(t,t) > 0$; else it is* **inactive** *at t. An interval $[s,t]$ is an* **active period** *for f_i starting at s, if f_i is inactive at time $s-1$ and is continuously active at all times in the interval $[s,t]$.*

DEFINITION 3. *Let $t = 0$ be the start of a system busy period. Let a and b, $a < b$, be the start times of two successive active periods for f_i. Let $R_i(0,b)$ be the cumulative arrivals for f_i in the interval $[0,b]$. The* **Arrival Upper Bound** *$\mathscr{U}_i^b(t)$ for the active period $[b,t]$ of f_i is defined as: $\mathscr{U}_i^b(t) = min\{\mathscr{U}_i^a(t),\ R_i(0,b) + \sigma_i + \rho_i(t-b)\}$.*

The definition of the AUB function for flow f_i is illustrated in Figure 3(a). The arrival function $R_i(0,t)$ that represents the cumulative arrivals of f_i, is shown by the dotted line. The AUB consists of three segments $U_i^0(t)$, $U_i^a(t)$ and $U_i^b(t)$ corresponding to the start of the three active intervals $[0,p]$, $[a,r]$ and $[b,\infty)$. The AUB in the interval $[0,p]$ is given by $\mathscr{U}_i^0(t) = \sigma_i + \rho_i t$, and is shown by the solid line of slope ρ_i at $t = 0$. For $t \geq a$, since $R_i(0,a) + \sigma_i > U_i^0(a)$, we have $U_i^a(t) = U_i^0(t)$. The next active period for f_i begins at time b. Since $R_i(0,b) + \sigma_i < U_i^a(b)$, we have for $t \geq b$, $\mathscr{U}_i^b(t) = R_i(0,b) + \sigma_i + \rho_i(t-b)$. A *well-behaved flow* will never send more requests than specified by its current AUB function.

LEMMA 1. *Let a be the start of the active period for f_i that includes time t. Then f_i is* **well behaved** *at t if and only if $R_i(0,t) \leq \mathscr{U}_i^a(t)$.*



(a) Computation of AUB (b) Compute tags using AUB

Figure 3: (a) Computation of Arrival Upper Bound (AUB) function for a flow f_i (b) Using AUB function to compute tags. Well behaved arrival at t_1 gets a deadline of $t_1 + \delta_i$; others are delayed based on how much the arrivals exceed the AUB

System Capacity: Service guarantees for well behaved flows can only be met if the system has sufficient capacity to meet their stipulated requirements. A lower-bound on the system capacity, referred to as the **System Capacity Constraint** is defined below. The constraint is defined based on the initial AUB functions $U_i^0(t)$ of the flows, and their latency bounds δ_i. Note that the constraint depends only on the static parameters of the flow requirements, and not the dynamic behavior of the schedule. Hence it can be easily checked to determine eligibility of a flow for admission control purposes.

To illustrate the definition we use an example of two flows with requirements f_A = ⟨50 IOs, 50 IOPS, 200 ms⟩ and f_B = ⟨110 IOs,

100 IOPS, 600 ms⟩. Consider the case when the two flows begin at $t = 0$; both send their maximum burst and then send at a fixed rate corresponding to their throughput requirements. The first deadline is at 200ms for f_A and the system must service 50 IOs requested at $t = 0$ by f_A by this time. This provides a lower bound on the capacity of 250 IOPS. At $t = 600$ ms, all requests of f_A received before $t = 400$ms must be serviced, as must all 110 IOs in the initial burst of f_B. Hence the system must service at least $\sigma_A + \rho_A(\delta_B - \delta_A) + \sigma_B = 50 + 50 \times 400/1000 + 110 = 180$ IOs by $t = 600$ ms, requiring a capacity of at least 300 IOPS. Beyond $t = 600$ ms, the system must service all requests from f_A and f_B that arrive at rates ρ_1 and ρ_2 respectively. Hence $C \geq 50 + 100 = 150$ IOPS. The maximum of these is a lower bound on system capacity, and in this example is 300 IOPS.

This motivates the definition below. By considering the case of all m flows sending their bursts at $t = 0$, followed by requests at their designated throughput it can be seen, as in the above example, that meeting this constraint is necessary if the system is to deterministically guarantee the requirements of all the flows.

DEFINITION 4. *Let the flows be arranged in order of non decreasing latencies, represented by $\delta_1 \leq \delta_2 \leq \cdots \leq \delta_m$. Let C denote the system capacity. The* **System Capacity Constraint** *is defined by the following equations:*

$$\sum_{\forall i} \rho_i \leq C \tag{1}$$

$$\forall\, k\ , \sum_{i \leq k} \sigma_i + \sum_{i=1}^{k} \rho_i(\delta_k - \delta_i) \leq C \times \delta_k \tag{2}$$

LEMMA 2. *A necessary condition required to guarantee that all well-behaved flows meet their deadlines is for the capacity C to meet the System Capacity Constraint.*

In Section 4, we show that these conditions are *sufficient* to guarantee that flows never miss their deadlines. This does not obviously follow from the definition. For instance if f_A in the above example sends its burst at the same time as f_B is sending at its steady rate of 100 IOPS, then to meet the burst of 50 IOs of f_A within the 200 ms deadline and serve the requests of f_B (100 IOPS) may appear to require a capacity of 350 IOPS. However, as we show a capacity of 300 IOPS is sufficient to meet all deadlines.

3.2 *p*Clock Algorithm

The algorithm assigns tags to arriving requests. Each request receives a *start tag* and a *finish tag*. The scheduler dispatches the request with the smallest finish tag to the server on a request completion. The tags assigned are controlled by the current arrival upper bound (AUB) function of the flow. Informally, requests that are within their arrival bounds will be assigned finish tags equal to the real-time deadlines of the requests, while requests that exceed the arrival bounds are assigned higher deadlines. Hence well-behaved flows will have their requests serviced in preference to those that exceed their arrival constraints, and will meet their latency bounds if they are serviced by their finish tags.

Algorithm 1 provides a high level description of the actions of the scheduler at request arrival and scheduling instants. There are three actions to be performed when a request arrives: *UpdateNumtokens* updates the AUB function for the present arrival time t, *CheckandAdjustTags* is used to resynchronize flows thereby avoiding starvation, and *ComputeTags* assigns start and finish tags as indicated above, based on the AUB. We describe these in more detail below. A formal description of various components is presented in Algorithm 2 along with related notation in Table 2.

Symbol	Meaning
S_i^r	Start tag of request r of f_i
F_i^r	Finish tag of request r of f_i
$MinS_i$	Minimum start tag of a pending request from f_i
$MaxS_i$	$1/\rho_i$ + Maximum start tag of a pending request from f_i
$numtokens_i$	Number of tokens available for f_i

Table 2: Symbols used and their descriptions

1 **Request Arrival:**
2 Let t be arrival time, of request r from f_i;
3 UpdateNumtokens( );
4 CheckandAdjustTags( );
5 ComputeTags( );

1 **Request Scheduling:**
2 Choose the request w with minimum finish tag F_j^w and dispatch to the server;
3 Let the chosen request be from flow f_k with *start tag* S_k^w;
4 $MinS_k = S_k$;

Algorithm 1: *p*Clock algorithm

UpdateNumtokens:
On each request arrival from flow f_i: Let Δ be time difference between the current time and the time of the previous request of f_i;
$numtokens_i$ += $\Delta \times \rho_i$;
if *($numtokens_i > \sigma_i$)* **then**
  $numtokens_i = \sigma_i$

CheckAndAdjustTags:
Let C be the set of currently busy flows;
if *($\forall\, j \in C$, $MinS_j > t_r$)* **then**
  mindrift = $min_{j \in C}\ \{MinS_j - t_r\}$;
  $\forall$ j $\in$ C, Subtract *mindrift* from $MinS_j$, $MaxS_j$ and all *start* and *finish* tags

ComputeTags:
if *($numtokens_i < 1$)* **then**
  $S_i^r = \max\{MaxS_i,\ t\}$;
  $MaxS_i = S_i^r + 1/\rho_i$
else
  $S_i^r = t$;
$F_i^r = S_i^r + \delta_i$;
$numtokens_i = numtokens_i - 1$;

Algorithm 2: Components of *p*Clock algorithm

UpdateNumtokens: In order to assign tags, the scheduler must first update the arrival upper bound function $\mathscr{U}_i^a()$ to the current time t. It maintains a variable $numtokens_i$ for each flow f_i. The variable keeps track of the difference between the AUB at time t and the cumulative number of arrivals up to that time (*i.e.* $\mathscr{U}_i^a(t) - R_i(0,t)$). Its value indicates the number of requests that can be made by f_i at t without violating the arrival constraints. Hence a value less than one means that a well behaved flow cannot make any request at t.

The initial value of $numtokens_i$ is set to σ_i at the start of a system busy period. It is decremented by 1 every time a request from f_i arrives, and continuously increases at the rate of ρ_i IOPS. Hence in an interval of Δ seconds it will be incremented by $\Delta \times \rho_i$, but the value is capped at σ_i.

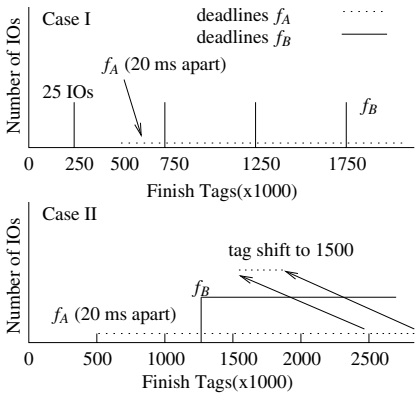


Figure 4: Finish tags assigned by *p*Clock for two cases

Compute Tags: This routine assigns *start* and *finish* tags (S_i^r and F_i^r respectively) to the request r from f_i arriving at time t. F_i^r is simply set to the sum of S_i^r and the latency bound δ_i. The value assigned to the start tag S_i^r depends on whether the request is within the AUB or exceeds it. In the first case (recognized by $numtokens_i \geq 1$), S_i^r is set to the current time t. If the total number of requests made by f_i through time t exceeds AUB ($numtokens_i < 1$), the start tag will be assigned a future time greater than t. In particular the start tag is set roughly to the time it would have taken a well behaved flow (with identical burst and throughput specifications) to send the same number of requests.

Figure 3(b) shows how the tags are set. At t_1 the total number of requests made by f_i is below the arrival upper bound. Hence the request is assigned a start tag of t_1 and a finish tag of $t_1 + \delta_i$. On the other hand, at t_2 the total requests $R_i(0,t_2)$ exceeds the AUB. Hence the start tag is set to t_a, the earliest time at which $R_i(0,t_2)$ does not exceed the AUB.

The routine maintains two variables $MinS_i$ and $MaxS_i$ for each flow f_i. The start tag of the last request of f_i dispatched to the server is maintained by the variable $MinS_i$. Requests belonging to f_i will have start tags spaced $1/\rho_i$ apart, *i.e.* $MinS_i + 1/\rho_i$, $MinS_i + 2/\rho_i$, $\cdots$, $MaxS_i - 1/\rho_i$. Hence if there is another request from f_i while this backlog persists it will be assigned a start tag of $MaxS_i$, and $MaxS_i$ will be increased by $1/\rho_i$.

Example 1: We illustrate the assignment of tags and how the algorithm meets deadlines of bursty flows with an example. Consider two flows f_A and f_B with requirements $\langle$0, 50 IOPS, 500 ms$\rangle$ and $\langle$25 IOs, 50 IOPS, 250 ms$\rangle$. The system capacity is assumed to be 100 IOPS. f_A sends requests at a uniform rate of 50 IOPS, whereas f_B sends a burst of 25 IOPS every 0.5 sec. The tags will be assigned as described below. (For convenience we scale values by 1000 *i.e.* we consider throughput in IOs per millisecond and assign all tags in milliseconds). f_A will have start tags of 0, 20, 40, 60, $\cdots$ (0, $1000/\rho_A$, $2000/\rho_A$ etc.) and finish tags starting from 500 and spaced at intervals of 20 ms each. f_B will have 25 start tags ($\sigma_a = 25$) with value 0 whose corresponding finish tags are all 250. This will repeat again at 500 ms where the burst of 25 requests will be assigned finish tags of 750. The finish tags are shown as Case I in Figure 4. The scheduling based on finish tags will complete the entire first burst from f_B before doing any request of f_A; similarly it will do the entire second burst (with finish tag 750) after only 12 requests of f_A. This is not possible using previous techniques for bandwidth allocation which would interleave the requests from the two flows (since they have the same throughput requirements) resulting in large latencies for f_B.

We finally discuss the remaining component of the algorithm. **Adjust Tags:** The motivation for this component is to allow flows to flexibly use spare capacity without being penalized. For instance a closed-loop client f_i that keeps a fixed number of requests outstanding at the server, will automatically increase its input rate when the system is lightly loaded and slow down as the load on the system increases. Without the tag adjustment component, the scheduler could penalize f_i for the extra service, because the algorithm would set the start tags of f_i's requests far into the future beyond the current time t. As a new flow f_j gets activated it will receive start tags beginning from the current time t. Hence f_i will get starved till the tags of f_j catch up. We need a mechanism to synchronize flows like f_i with newly activated flows.

The routine checks for the state when all the start tags of all requests in the system are greater than the current time t. This indicates that busy flows have received more than their guaranteed amount of service by t. At this time, rather than allowing the tags to keep running ahead of real time, the algorithm recomputes the tags so that the smallest start tag after readjustment coincides with the current time t. The relative values of the tags are not changed and they just shift as a block by an offset that depends on the difference between the smallest start tag in the system and the current time . Since new flows will begin their tagging from the current time as well, all flows compete fairly from this point on, avoiding starvation. The time at which the tags are adjusted is also called a *synchronization instant*.

Note that this shift raises the possibility that some flows may now miss their deadlines since a greater number of requests than anticipated are pushed into a given time interval. However, as we show in Theorem 1 the readjustment of tags does not result in any missed deadlines. Hence this provides a simple method to avoid starvation, while still maintaining the ability to handle bursts and meet latency bounds.

Example II: We present an example to illustrate how the tags are adjusted. Consider the same flows f_A and f_B as example 1 but with a different arrival pattern (see Case II of Figure 4). Let f_A send 100 IOs at $t = 0$ and 50 IOs from $t = 1$ onwards. Let f_B send a burst of size 25 IOs at $t = 1$ and 50 IOPS onwards. The finish tags for f_A will go from 500 ms to 2500 ms spaced by 20 ms each. At $t = 1$, all the 100 IOs from f_A have completed. f_B will have 25 finish tags at 1250 ms and the remaining tags will be 1270, 1290 and so on (again spaced by 20ms each). Now if no tag shifting is done, f_A will get starved till the tags of f_B catch up to 2500 ms. To avoid this behavior *p*Clock will shift the minimum start tag of f_A from 2000ms to 1000 ms (and the corresponding finish tag from 2500 to 1500) when the first arrival from f_B occurs. Thus f_A will start competing with f_B as soon as the finish tag of f_B reaches 1500 ms.

3.3 Calculation of Spare Capacity

DEFINITION 5. *The* **spare capacity** *of the system is the maximum amount of service that can be provided to a background job while ensuring that every well-behaved flow f_i meets its deadlines.*

The spare capacity can be allocated either to the foreground (contractual) flows, so that they can get more than the contracted amount of service or to some background job. The first case is easy to implement since one can schedule the background job only when there is no request present from any of the foreground flows. However a potential drawback of this scheme is that the requests of the background job may be done in a scattered manner rather than in a batch.

Allocation of all spare capacity to a background job is much more challenging, because we need to identify the amount of spare capacity that can be given to it without hurting any of the well-

behaved flows. As shown in Lemma 5 whenever the tags are adjusted by the algorithm, the system can provide a burst of size $\sigma_i - 1$ from each of the busy flows to the background job. We allocate the combined burst from all busy flows to the background task. Let F be the set of busy flows; their tags would have been adjusted based on the current time t. The background requests will be scheduled as follows: $\forall f_i \in F$, schedule $\sigma_i - 1$ requests of the background job with deadline equal to $t + \delta_i$. This has two major benefits over existing methods. First, the guarantees to well behaved flows are not missed and second, the requests of the background job are done in batches, which can lead to better disk utilization as many background jobs (backup, defragmenter) tend to be highly sequential.

Note that bandwidth allocation schemes based on virtual tags, such as WFQ, SFQ, WF^2Q etc. can only implement the first case, where a background job is scheduled when no one else is active. This is because there is no relation between virtual tags and actual deadlines. In our case one can tune the allocation of spare capacity between contractual flows and background jobs by assigning bursts to the background job based on the actual slack time between the minimum start tag and the current time.

4. PROOF OF CORRECTNESS

In this section we provide a proof of the scheduling guarantees provided by our algorithm. We will show that if the system capacity satisfies the constraint described in Section 3 and all flows are well-behaved then all flows will meet their deadlines. We will then show the stronger result that well-behaved flows are insulated from the behavior of badly behaved flows by proving that the deadlines are met for all the well-behaved flows even if other flows exceed their AUB curve.

LEMMA 3. *Let f_i be a flow that is well-behaved. If every request of f_i completes service before the finish tag assigned to it by pClock, then f_i never misses a deadline.*

LEMMA 4. *Let u be the last time before t at which the scheduler adjusts the tags of the set of busy flows. The total number of requests (from all flows) with finish tags between u and t that have not yet been serviced by u, is upper bounded by $C \times (t-u)$.*

LEMMA 5. *Let $\mathscr{A}$ denote the set of busy flows at a synchronization instant u at which the tags are adjusted. Then the system has spare capacity at least $\sum_{j \in \mathscr{A}} (\sigma_j - 1)$ at time u.*

As a consequence of Lemma 5, at a synchronization instant u, the system can schedule a background job in bursts of $\sigma_j - 1$ requests with deadline $u + \delta_j$ for each $f_j \in \mathscr{A}$, without affecting the foreground flows.

LEMMA 6. *Let s be the start of the system busy period that contains time t. The total number of requests (from all flows) with finish tags between s and t is upper bounded by $C \times (t-s)$.*

THEOREM 1. *If a flow f_i is well behaved and the system capacity constraint is satisfied, then f_i never misses its deadline, irrespective of the behavior of other flows.*

5. PERFORMANCE EVALUATION

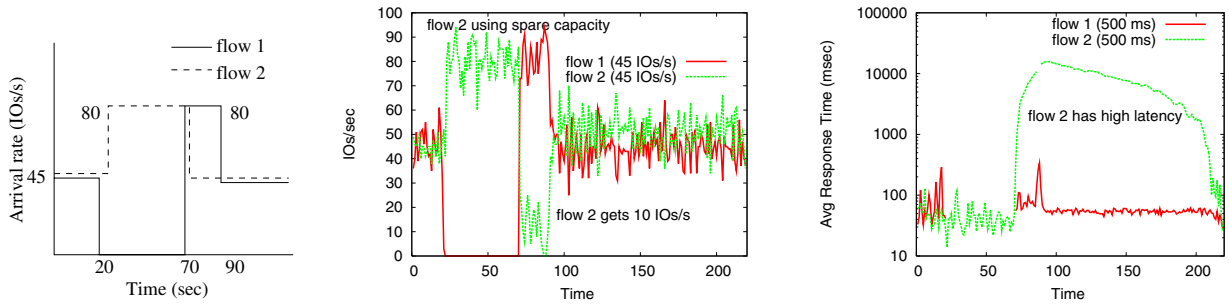
In this section we present extensive results of evaluating *p*Clock using simulation and actual implementation in Linux kernel 2.6, on synthetically generated workloads and various benchmarks. We evaluated our approach in three scenarios. First, we simulated a single disk system using DiskSim3.0 [2], mainly to show comparison between *p*Clock and other methods. This is a simple environment that is easy to manage, understand, and reason about. Next, we simulated a storage system using DiskSim3.0. The storage system consists of 16 Seagate ST39102LW disks each with approximately 9GB capacity arranged in 4 strings of 4 disks each. Each string is configured as a JBOD. The storage server consists of a device driver with four attached controllers (one per string). Our algorithm *p*Clock is placed at the driver. Each disk runs a cyclic SSTF (shortest seek time first) disk-head scheduling algorithm and has a queue of size 4 for command queuing. DiskSim models many of the details of the storage system such as caching, bus-conflicts, write buffering, and low-level scheduling at disks that are abstracted away in our model and analysis; this allows us to confirm that our algorithm works in a storage environment with these features. Flows have a mixture of 70% reads and 30% writes unless otherwise stated. Finally, we implemented our algorithm in the Linux kernel and tested it with various benchmarking tools. The algorithm is implemented as a module[1] that creates multiple pseudo devices, one per flow, all backed up by a single disk. Each pseudo device is associated with bandwidth, latency and burst requirements and represents a single flow. All requests going to the disk go through pseudo devices and our scheduler decides which requests to forward and when. We experiment with both *noop* and *anticipatory* as our underlying scheduler in the kernel; *noop* is a very simple scheduler that just sends the requests to device in FCFS order, and *anticipatory* [14] scheduler introduces some delay before scheduling a request in anticipation of a request closer to the one that is just serviced.

We aim to show the following properties of our framework with this evaluation: (i) additional features of our solution by comparison with other existing algorithms, specifically the SCED algorithm [23] and WF^2Q+ [3, 4]; (ii) the ability to isolate performance of well and ill-behaved flows; (iii) ability to handle bursts in arrivals; (iv) ability to assign available spare capacity without hurting the deadlines of well-behaved flows.

5.1 Comparison with SCED and WF^2Q+

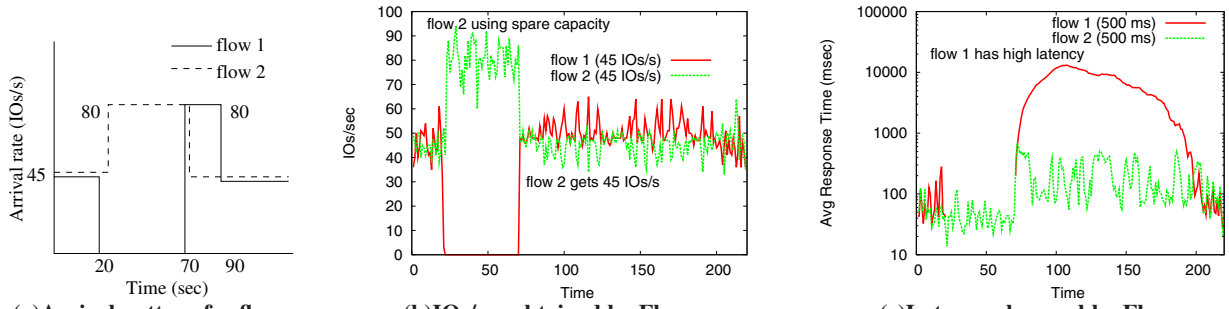
Since SCED and WF^2Q+ are the existing algorithms that come closest to matching the features of *p*Clock, we chose to compare the performance of *p*Clock against these two algorithms. We implemented both SCED and WF^2Q algorithms in DiskSim and used a single disk server for comparison. As noted in Section 2.1, if a flow uses spare capacity resulting from idle periods of other flows, then SCED may starve the flow when the idle flows become active. We show this behavior using the following test case: 2 identical flows with requirements $\langle$10 IOs, 45 IOPS, 500 ms$\rangle$ each. The system capacity is approximately 95 IOs/sec, which we found by sending random IOs to the disk. As shown in Figure 5(a), flow 1 sends 45 IOs/sec between $t = 0$ and $t = 20$ seconds, then is inactive until $t = 70$ seconds and then sends around 80 IOs/sec until $t = 90$ seconds. Flow 2 sends 45 IOs/sec until $t = 21$ seconds, then 80 IOs/sec between $t = 22$ and $t = 71$ seconds, and returns to 45 IOs/sec from $t = 71$ onwards. This simulates the behavior of an application that tries to use the spare capacity by sending a larger number of requests when it detects that there is spare capacity at the storage server. The application sends more requests if deadlines are not missed and goes back to its contractual rate if the latency starts to increase. We have shown the average arrival rate; the actual inter-arrival time follows an exponential distribution. Figure 5(b) shows the bandwidth allocated by SCED and Figure 5(c) shows the latency observed. Figure 6 shows the arrival pattern, bandwidth allocated, and latency observed using *p*Clock. Note that when flow 1 sends requests at a higher arrival rate after $t = 70$ seconds, flow

[1]We thank Richard Golding and Theodore Wong of IBM Almaden for providing the skeleton module for creating psuedo devices.



(a)Arrival pattern for flows **(b)IOs/sec obtained by Flows** **(c)Latency observed by Flows**

Figure 5: SCED Algorithm: Flow 2 uses spare capacity from 20 to 70 seconds and gets penalized when flow 1 gets active at 70 seconds. Flow 2 is starved and it only gets about 10 IOs/s, which is the capacity left after servicing flow 1



(a)Arrival pattern for flows **(b)IOs/sec obtained by Flows** **(c)Latency observed by Flows**

Figure 6: *p*Clock Algorithm: Flow 2 uses spare capacity from 20 to 70 seconds. Later at 70 seconds, when flow 1 sends extra arrivals, flow 2 still meets its requirements and flow 1 suffers. The spike in flow 2's latency is because it reduces it arrival rate at 71 sec

2's throughput falls to 10 IOs/sec in the case of SCED. However flow 2 gets 45 IOs/sec after $t = 70$ in the case of *p*Clock. Similarly, the latency of flow 2 in *p*Clock rises briefly to about 690ms when flow 1 becomes active, and then drops back to normal within 2 seconds. However in SCED the latency for flow 2 goes up to almost 15 sec, and it takes almost 135 sec for it to come back to its guaranteed value. In practice an application may abort and declare IO failure when such large latencies are encountered. This shows that SCED doesn't support viable spare capacity usage, and can lead to starvation.

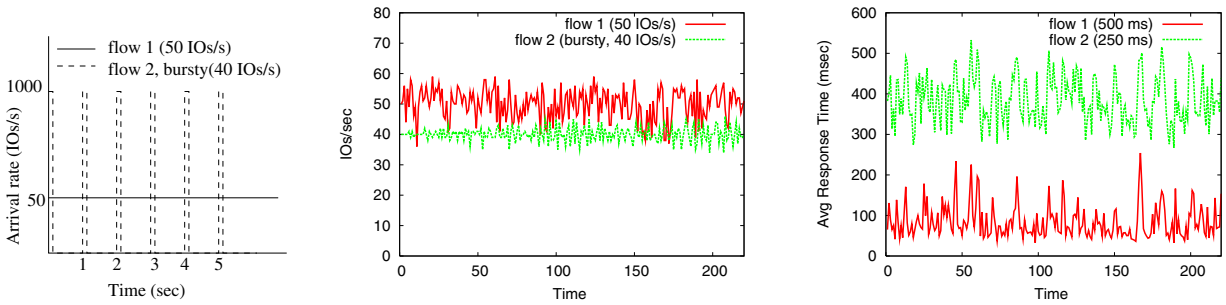
In the case of virtual time based algorithms, WF^2Q+ has been shown analytically to have the best latency bounds and an efficient implementation [4]. All these schemes are primarily scheduling algorithms for bandwidth allocation, and there is no mechanism for reducing latency or handling bursts other than increasing the bandwidth allocation to a flow. To demonstrate and contrast with this behavior, we experimented with the following setup: 2 flows with requirements $\langle$5 IOs, 50 IOPS, 500ms $\rangle$ and $\langle$20 IOs, 40 IOPS, 250 ms $\rangle$ respectively. The system capacity is around 95 IOs/sec. Flow 1 sends 50 IOs/sec and flow 2 has periodic bursty arrivals where it sends 40 IOs at the beginning of every 1-second window (in first 40 ms). Figure 7 shows the arrival pattern, bandwidth allocated and latency observed by the WF^2Q+ scheduler, and Figure 8 shows the arrival pattern, bandwidth allocated and latency observed using *p*Clock. Note that, in the case of WF^2Q+ the deadlines of flow 2 are missed because of the bursty arrivals, but the throughput guarantees are always met. In the case of *p*Clock, flow 2 never misses its deadlines, which also implies that its bandwidth guarantees are fulfilled. This is because, unlike WF^2Q+, burst handling is built into *p*Clock.

5.2 Performance Isolation

One of the basic requirements of our framework is to provide performance isolation, *i.e.,* to prevent an ill-behaving flow from affecting the throughput and response times of other well-behaved flows. To test if *p*Clock can effectively provide isolation among the flows, we experimented with three flows with requirements: $\langle$100 IOs, 650 IOPS, 500 ms $\rangle$, $\langle$50 IOs, 400 IOPS, 300 ms $\rangle$ and $\langle$50 IOs, 250 IOPS, 200 ms$\rangle$ respectively. The storage system capacity is around 1500 IOs/sec, obtained by sending random requests to the system, which has 16 disks. (All the DiskSim experiments from here on use the 16-disk storage system.)

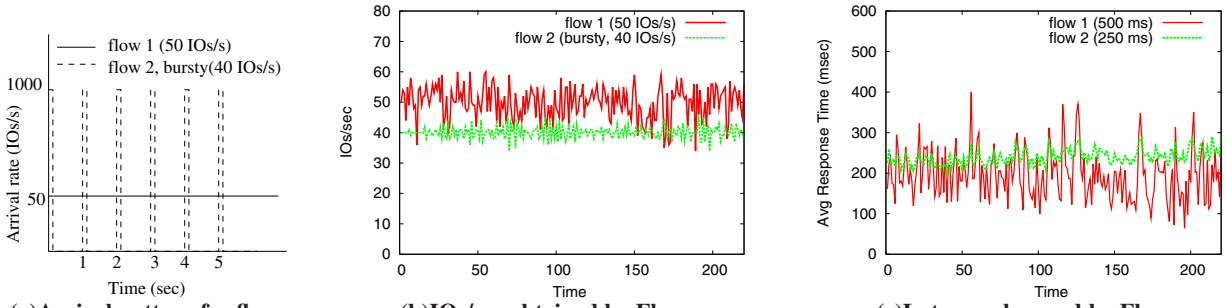
In this experiment, two of the flows are ill-behaved because they exceed their request rate specification. Both flows 1 and 3 start at their specified request rate, but increase the arrival rate by 30 IOs/sec and 20 IOs/sec respectively every 10 seconds (shown in Figure 9(a)). The results are shown in Figures 9(b) and 9(c). The well-behaved flow 2 never misses any deadlines despite the large number of arrivals from other flows. Also note that flow 3's latencies rise much faster than those for flow 1 because the proportionate increase in the arrival rate for flow 3 (about 8%) is higher compared to flow 1 (about 4.6%).

Next we explored bad behavior in terms of large bursts in the arrival pattern and show that *p*Clock still meets all guarantees. A bursty arrival pattern is simulated with ON-OFF arrivals, while keeping the average arrival rate constant. A bursty flow is more likely to miss its deadline because of higher queuing delay. However, the algorithm should be able to absorb bounded bursts. To test the effect of sudden bursts we used the following flow parameters: $\langle$50 IOs, 650 IOPS, 400 ms $\rangle$, $\langle$200 IOs, 400 IOPS, 200 ms $\rangle$ and $\langle$50 IOs, 330 IOPS, 200 ms$\rangle$ respectively. Flow 1 sends a burst of size 650 every second in single spike. A spike of n IOs is simulated



(a)Arrival pattern for flows (b)IOs/sec obtained by Flows (c)Latency observed by Flows

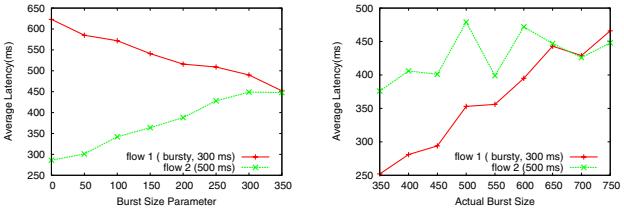
Figure 7: WF^2Q Algorithm: Flow 2 sends a burst of 40 IOs every second (IOs are sent in first 40ms interval) and continuously misses its deadlines. This is because it is not designed to handle bursty behavior



(a)Arrival pattern for flows (b)IOs/sec obtained by Flows (c)Latency observed by Flows

Figure 8: pClock Algorithm: Flow 2 sends a burst of 40 IOs every second and observes latency close to its deadline

by sending n IOs at a gap of 0.1 ms each. Thus, a spike of size 100 IOs will be sent in a period of 10 ms. Flow 2 sends a spike of 200 IOs every 500ms, thereby sending total arrivals of 400 IOs/sec. The arrival pattern is shown in figure 10(a). Bandwidth and latency observed by various flows are shown in Figures 10(b) and 10(c). Note that flows 1 misses its deadlines because it sends bursts of much larger sizes (larger than its σ value) but this doesn't affect the deadlines of flow 2 and 3 in any way. Flow 2 is able to meet its deadlines inspite of the burst because its bursts are within its burst size parameter of 200. We note here that if the flows are not bursty then all the deadlines are actually met (since the system is capable of doing it) and the deadlines are missed only because of the large burst size.



(a) Average Latency vs σ (b) Average latency vs actual bursts

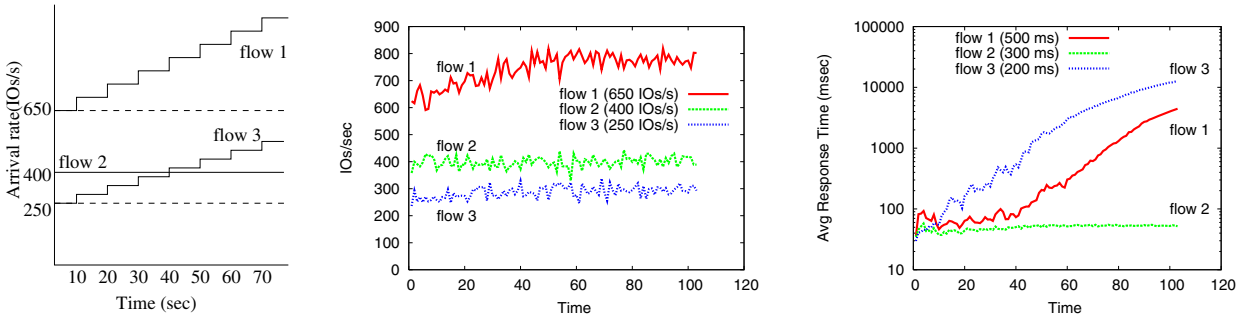
Figure 11: Latency observed by flows when we vary either burst parameter for flow 1 or actual burst size. Throughput is kept constant in these tests.

5.3 Handling Bursty Arrivals

In this section we will show the effectiveness and sensitivity of pClock in the presence of bursty arrivals (spikes). Our algorithm is able to absorb bounded spikes. The spike size that can be absorbed without missing any deadlines depends on the system capacity and requirements of other flows. We will show two characteristics of our burst handling mechanism: first we show that as we vary the burst parameter of a flow, pClock is able to steal latency from one flow to satisfy the burst of another flow. Second we show that as long as the size of the spike is less than the burst parameter, pClock never misses the deadline. As the spike size increases, the latency of the bursty flow increases but the other flow never misses its deadline. This will also show the sensitivity of response time with the variation in burst parameter of the flow.

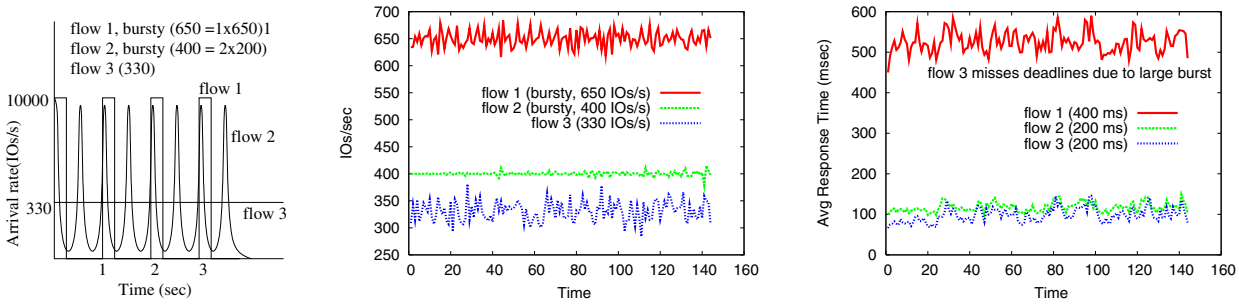
We experimented with 2 flows with following parameters: $\langle$ 350 IOs, 750 IOPS, 300 ms $\rangle$, $\langle$50 IOs, 750 IOPS, 500 ms $\rangle$. Flow 1 sends a burst of size 750 IOs (single spike) and we decrease the burst parameter (σ_1) for flow 1 from 350 to 0. Figure 11(a) shows the average latency obtained by each of the flows in presence of flow 1 bursts. Note that the latency for flow 1 increases as we decrease σ_1 and the latency for flow 2 decreases with the decrease in σ_1. Thus, our algorithm is able to steal latency from one flow and give it to another with variation in the burst size parameter. Note that in this case flow 1 misses its deadline because it is sending a larger burst that the limit but flow 2 never misses its deadline.

Next we will show that pClock is able to absorb the bursts if they are not higher than the burst size bound (σ). The value of σ cannot be arbitrarily increased because of the system capacity constraint. Figure 11(b) shows the variation of average response time observed by flows when we keep $\sigma_1 = 350$, but we vary the spike size sent by flow 1 between 350 and 750 IOs. The inter spike gap is decided such that the IO rate is still 750 IOs/s. Note that flow 1 is able to meet its deadlines on average until a burst size of 450, after that its average latency goes to 350ms, whereas the desired deadline is 300ms.



(a)Arrival pattern for flows **(b)IOs/sec obtained by Flows** **(c)Latency observed by Flows**

Figure 9: Throughput and latency observed by flows, when flows 1 and 3 are increasing its arrival rate by 30 IOs/sec and 20 IOs/sec every 10 seconds respectively. Flow 2 is unaffected by the behavior and it never misses its deadline



(a)Arrival pattern for flows **(b)IOs/sec obtained by Flows** **(c)Latency observed by Flows**

Figure 10: Throughput and latency observed by flows, when arrivals of flows 1 and 2 are bursty. Flow 1 sends a spike of 650 IOs/sec. Flow 2 sends two spikes of 200 IOs every 500ms. Flow 2 and 3 never miss their deadlines because flow 2 is sending bounded bursts

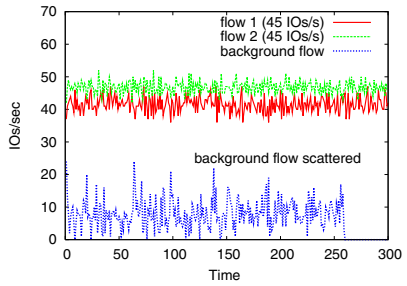
5.4 Spare Capacity Allocation

As mentioned before, we can allocate spare capacity in two ways: the first is to give all the spare capacity to contractual flows if they send more requests, and the second is to give it to background activities and provide only the guaranteed capacity to the contractual flows. Both policies can be useful under different scenarios.

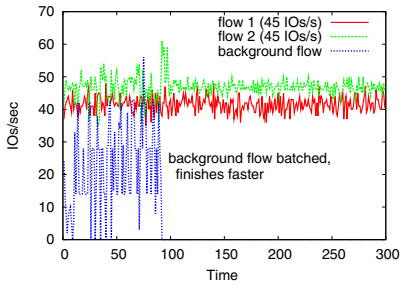
To test the behavior of these two policies, we used the following experimental set up: two foreground flows and one background flow with a high probability (0.95) of sequential requests. All three flows access a single disk. The background job has a fixed set of requests (2000 IOs in our experiment) that it needs to finish. We make flows send more than the specified request rates and see how long it takes for background job to complete. This simulates the behavior of a backup or defragmenter on the same disk as flows; we want the backup to finish as fast as possible without affecting the foreground tasks. The contractual parameters for both flows are $\langle$15 IOs, 45 IOPS, 500 ms $\rangle$. The system capacity for random IOs is around 95 IOs/sec. Flow 1 sends around 42 IOs/sec and flow 2 around 48 IOs/sec. The inter-arrival pattern for requests follows a uniform distribution between a minimum and a maximum value. Figure 12(a) shows the throughput obtained by flows when spare capacity is allocated to foreground flows and Figure 12(b) shows the bandwidth obtained by flows when the spare capacity is allocated to the background task. In the first case, the background task only gets a few requests when there is no other request pending in the system, whereas in the second case the background task is scheduled in batches whenever spare capacity is available. Note that the background task finishes mush faster with the second policy because of its sequential nature and because its requests are scheduled in batches, which leads to a higher disk throughput. Also the bandwidth of foreground tasks isn't affected and they still get the contractual values. The latency of flow 1 is also unaffected because it is well behaved, whereas flow 2 sees higher latencies because it is sending more arrivals than specified, and the spare capacity is given to background job. Thus, we can expedite a background job when needed while still providing contractual guarantees to other flows. A further improvement would be to allow a flexible way to share the spare capacity between foreground (contractual) flows and background jobs. This is outside the scope of this paper and is part of our future work.

5.5 Linux Implementation

We implemented *p*Clock in the Linux kernel as a loadable module. Once the module is loaded it creates pseudo devices. Each pseudo device can be backed up by a physical block device by calling a special mount operation that uses a specific `ioctl call` in the module. Each of the pseudo devices can be assigned three parameters - burst size, bandwidth, and latency by using `ioctl calls` defined in the module. Once this is done, we can run one application per pseudo device and observe the desired allocation from the algorithm. For these experiments we created four pseudo devices, all of them are backed by a common hard disk. This disk is only accessed by our module and the operating system is supported using another disk. The capacity of the disk is about 85 IOs/s. This is measured by sending random IOs to the device. The whole module is implemented using about 900 lines of C code, along with about 200 lines of code for mounting and unmounting operations. The actual algorithm code is only 60 lines and the rest is code for initialization, mounting and other setup related operations. First we will provide some results for various micro benchmarks that show the bandwidth allocation and latency control of *p*Clock. We implemented a workload generator for the micro benchmarks, using



(a) Case I: spare capacity to flows



(b) Case II: spare capacity to background job

Figure 12: IOs observed by contractual flows and background job, with two policies of implementing spare capacity. Note that when spare capacity is allocated to the background job, it finishes much faster (in 96 seconds) and disk throughput is much higher

Workloads	Specs (σ_i,ρ_i,δ_i)	ops/s	Avg B/W (mb/s)	Avg Latency (ms)
I (Webserver1)	$\langle 5, 30, 500 \rangle$	16.6	1.3	2705.5
I (Webserver2)	$\langle 5, 60, 500 \rangle$	33.2	2.7	1279.2
II (Varmail1)	$\langle 5, 30, 500 \rangle$	53.4	1.2	966.3
II (Varmail2)	$\langle 5, 60, 500 \rangle$	149.2	1.9	331.7
III (oltp)	$\langle 20, 60, 250 \rangle$	60.9	0.1	367.2
III (Webserver)	$\langle 5, 60, 500 \rangle$	23.9	1.9	942.9

Table 3: Filebench Results

threads to maintain multiple outstanding IOs and the *nanosleep* call for rate control. Later, we present some results for macro testing of *p*Clock using filebench [18], a benchmark tool created by Sun.

Bandwidth allocation: We ran two applications with requirements $\langle$ 5 IOs, 40 IOPS, 100 ms $\rangle$ for both. Flow 1 is rate controlled and sends 40 IOs/s whereas flow 2 is always backlogged with 8 IOs pending. Figures 13(a) and 13(b) show the bandwidth obtained and latency observed by two flows. Flow 2 gets around 45 IOs/s thereby utilizing the spare capacity in the system. However, flow 1 gets its required 40 IOs/s at a latency of 30ms without being affected by flow 2. The latency of flow 2 (around 180ms) is higher because it is continuously backlogged. We also experimented with two continuously backlogged flows with bandwidths of 20 IOs/sec and 60 IOs/sec, keeping other factors constant. Since the flows are always backlogged, they obtain about 24 and 63 IOs/sec. The average latencies observed are 320ms and 130ms respectively. This shows that *p*Clock is able to maintain a bandwidth allocation in proportion to the requirements. The latencies observed are consistent with the expected correlation between bandwidth allocation and latency. The flow with higher bandwidth allocation observes a smaller latency. In the absence of *p*Clock, both applications get an equal share of the bandwidth and no requirements are enforced. (Plots omitted due to lack of space.)

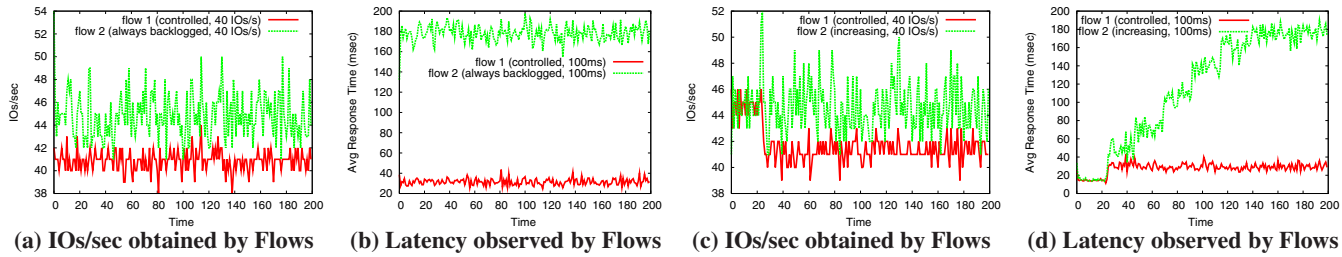
Latency handling: In order to show that *p*Clock is able to meet latency requirements, we experimented with 2 flows with requirements $\langle$ 5 IOs, 40 IOPS, 100 ms $\rangle$ for both. The first flow sends requests at the rate of 40 IOs/s whereas the second flow increases its rate after every 1024 IOs. Throughput and latency results are shown in Figures 13(c) and 13(d) respectively. We observe that the average latency for the first flow stays close to 30ms, whereas the other flow suffers a latency of 180ms at the peak (always backlogged with 8 requests). In terms of throughput, the first flow gets it share of 40 IOs/sec, whereas the other flow gets around 45 IOs/s, thereby utilizing the spare capacity without affecting flow 1.

Filebench Results: Filebench comes with a number of workloads intended to represent standard applications. We ran various workloads such as *webserver, varmail, oltp* using filebench. Each workload is associated with a pseudo device with an ext2 file system and it accesses files on that device only. For our experiments, each pseudo device is backed up by a partition on the hard disk. Thus, the workloads access different partitions on the same hard disk. We experimented with different requirements for devices and observed the impact of those reservations on actual workloads. Each workload has a set of parameters to tune the dataset and actual workload. We mostly used the default parameters in the workload files in filebench but modified certain fields such as filesize and the number of threads to ensure that the dataset is larger than the available memory. For example, the webserver workload uses the following parameters: 1000 files, file size = 256k, threads per workload = 16 and mean directory width = 20. We experimented with three scenarios, two concurrent webservers with bandwidth requirements in a 1:2 ratio, two varmail workloads with bandwidth requirements in a 1:2 ratio, and, finally, the oltp workload with webserver, such that both have similar bandwidth requirements, but oltp has a stringent latency requirement. The specifications and results reported by filebench are shown in Table 3.

Case I: Two webserver workloads with bandwidths in the ratio 1:2, while the other factors are same. The results show that the bandwidth reservation in a 1:2 ratio is visible at the application level. The average latency shown is for macro file operations such as *readfile, openfile, closefile, appendfilerand, deletefile* and thus do not correspond to block level latency. However, the average latency is inversely proportional to the bandwidth requirements. This was expected because the workloads are always backlogged and the disk is 100% utilized.

Case II: Two varmail workloads with bandwidths in the ratio 1:2, while the other factors are same. Again the bandwidth reservation of 1:1.6 is visible even at the varmail application level. The average latencies are in the ratio 3:1 and so are the number of ops/s. This is again an artifact of the averages computed by filebench over file level operations. This shows that providing a certain block-level guarantee can affect the application's performance in different ways depending on the file operations involved.

Case III: Finally, we experimented with two diverse workloads, oltp (which is rate controlled) and webserver (which is always backlogged). We set a stringent deadline and a higher burst size for the oltp workload. The results show that the average latency of oltp is much lower than that of the webserver workload. The bandwidth consumed by oltp is lower because oltp is rate controlled. Again, the ops/s shown are the file level operations and it is higher for oltp because oltp does a greater number of smaller operations, whereas webserver mainly does open-readwholefile-close on files.



(a) IOs/sec obtained by Flows **(b) Latency observed by Flows** **(c) IOs/sec obtained by Flows** **(d) Latency observed by Flows**

Figure 13: Linux Micro benchmarks: Plots (a) and (b) show the bandwidth allocation, where flow 1 is unaffected by flow 2, which is always backlogged. Plots (c) and (d) shows that flow 1 meets its bandwidth and latency requirements even when flow 2 increases its rate from 40 IOPS until it is always backlogged.

Overall, we conclude that setting bandwidth allocations at the block level does affect the application performance in the expected manner: a higher bandwidth allocation yields a higher operation rate at the application level. However, in order to control the operation rate of the application, an additional layer of control may be needed to set and adjust the resource allocations appropriately.

6. CONCLUSIONS

We presented the *p*Clock algorithm that allows multiple workloads to share storage, with each workload receiving the level of service it requires. *p*Clock allows each workload to specify its throughput, burst size and desired latency, and guarantees that the latency will be met so long as the workload does not exceed the specified burst size and IO rate specifications. *p*Clock has several other desirable features, such the ability to allocate spare capacity to the workloads or to background jobs. We show analytically that, under certain idealized assumptions, a well-behaved workload will never miss its deadlines. Since the assumptions made in modeling the problem may not always hold in practice, we have demonstrated using extensive simulation experiments and a real system implementation that our algorithm meets all of its goals. The algorithm is light-weight to implement and efficient to execute. In future work, we plan to explore how application-level QoS mechanisms interact with control of the storage service. We also plan to study how storage service QoS control interacts with low level scheduling, such as disk scheduling and command queueing in disks and disk arrays.

Acknowledgments

We thank Richard Golding and Theodore Wong from IBM Almaden Research Center for their generous help with the Linux implementation. The support of this research by the National Science Foundation under Grant CNS-0541369 is gratefully acknowledged.

7. REFERENCES

[1] Amazon simple storage service (amazon s3). http://www.amazon.com/gp/browse.html?node=16427261.

[2] The disksim simulation environment (version 3.0). http://www.pdl.cmu.edu/DiskSim/.

[3] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *INFOCOM (1)*, pages 120–128, 1996.

[4] J. C. R. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, 1997.

[5] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems, Volume 2*. IEEE Computer Society, 1999.

[6] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. In *Symposium on Reliable Distributed Systems*, pages 109–118, Oct 2003.

[7] R. L. Cruz. Quality of service guarantees in virtual circuit switched networks. *IEEE Journal on Selected Areas in Communications*, 13(6):1048–1056, 1995.

[8] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience*, 1(1):3–26, September 1990.

[9] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOMM'94*, pages 636–646, April 1994.

[10] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. Technical Report CS-TR-96-02, UT Austin, January 1996.

[11] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. Netw.*, 5(5):690–704, 1997.

[12] A. G. Greenberg and N. Madras. How fair is fair queuing. *J. ACM*, 39(3):568–598, 1992.

[13] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. In *SIGMETRICS '04/Performance '04*, pages 14–24, New York, NY, USA, 2004. ACM Press.

[14] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, Oct. 2001.

[15] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *SIGMETRICS '04/Performance '04*, pages 37–48, New York, NY, USA, 2004. ACM Press.

[16] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1(4):457–480, 2005.

[17] C. Lumb, A. Merchant, and G. Alvarez. Façade: Virtual storage devices with performance guarantees. *File and Storage technologies (FAST'03)*, pages 131–144, March 2003.

[18] R. McDougall. Filebench:application level file system benchmark. http://www.solarisinternals.com/si/tools/filebench/index.php.

[19] T. S. E. Ng, D. C. Stephens, I. Stoica, and H. Zhang. Supporting best-effort traffic with fair service curve. In *Measurement and Modeling of Computer Systems*, pages 218–219, 1999.

[20] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1(3):344–357, 1993.

[21] A. K. Parekh and R. G. Gallagher. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Trans. Netw.*, 2(2):137–150, 1994.

[22] A. L. N. Reddy, J. Wyllie, and K. B. R. Wijayaratne. Disk scheduling in a multimedia I/O system. *ACM Trans. Multimedia Comput. Commun. Appl.*, 1(1):37–59, 2005.

[23] H. Sariowan, R. L. Cruz, and G. C. Polyzos. Scheduling for quality of service guarantees via service curves. In *Proceedings of the International Conference on Computer Communications and Networks*, pages 512–520, 1995.

[24] P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. In *ACM SIGMETRICS*, pages 44–55. ACM Press, 1998.

[25] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking*, 6(5):611–624, 1998.

[26] I. Stoica, H. Zhang, and T. S. E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time, and priority services. *IEEE/ACM Trans. Netw.*, 8(2):185–199, 2000.

[27] S. Suri, G. Varghese, and G. Chandramenon. Leap forward virtual clock: A new fair queueing scheme with guaranteed delay and throughput fairness. In *INFOCOMM'97*, April 1997.

[28] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. In *MASCOTS*, pages 135–142, 2005.

[29] L. Zhang. VirtualClock: A new traffic control algorithm for packet-switched networks. *ACM Trans. Comput. Syst.*, 9(2):101–124.