Bandwidth Allocation in a Self-Managing Multimedia File Server *

Vijay Sundaram and **Prashant Shenoy**

Department of Computer Science,
University of Massachusetts,
Amherst, MA 01003.
{vijay,shenoy}@cs.umass.edu

ABSTRACT

In this paper, we argue that manageability of file servers is just as important, if not more, as performance. We focus on the design of a self-managing file server and address the specific problem of automating bandwidth allocation to application classes in single-disk and multi-disk servers. The bandwidth allocation techniques that we propose consists of two key components: a workload monitoring module that efficiently monitors the load in each application class and a bandwidth manager that uses these workload statistics to dynamically determine the allocation of each class. We evaluate the efficacy of our techniques via a simulation study and demonstrate that our techniques (i) exploit the semantics of each application class while determining their allocations, (ii) provide control over the time-scale of monitoring and allocation, and (iii) provide stable behavior even during transient overloads. Our comparison with a static allocation technique shows that dynamic bandwidth allocation can yield queue lengths that are 59% smaller during overloads and admit a larger number of soft real-time clients into the system.

1. Introduction

In this paper we focus on techniques for improving the *manageability* of multimedia file servers. Modern file servers store increasingly heterogeneous data and service workloads with diverse performance requirements. Concurrent to these trends, disk capacities are doubling every 18 months as dictated by Moore's law and are accompanied by a corresponding increase in the volume of data stored on file servers [10]. The growing heterogeneity of file server workloads and increasing storage capacities have made the task of managing modern file servers very complex. Studies have shown that the chances of a misconfigured or sub-optimally configured server are growing [4]. In a world where information is increasingly available online, the cost of such misconfigurations can be high since even a short down-time can result in substantial revenue losses. Thus, system administrators must deal with the difficult task of configuring large, complex file servers so as to achieve high availability and performance. Furthermore, reconfiguration and tuning of such servers is required on a continual basis to deal with long-term changes in the workload or incremental growth. These trends motivate the need for techniques to improve the *manageability of large file servers*. Others in the computing research community share this belief. In fact, it has been argued that the problems of maintainability, availability and growth of computing systems have overshadowed that of performance and that the traditional focus on performance is less important in today's environments [7, 13].

We are developing a self-managing file server to address these issues. A self-managing file server automates the tasks of configuring and tuning the server in order to achieve high availability and performance. Such a server can automatically react to long-term and short-term (transient) variations in the workload as well as failures and changes in the server configuration. Since human intervention is required only for unusual circumstances, this simplifies the administration of large servers, reduces administration costs, and most importantly, reduces the chances of human error. Designing a self-managing file server introduces several research challenges such as the design of workload monitoring techniques, self-managing policies for placement and retrieval, techniques for handling failures and incremental growth.

In this paper, we focus on the specific problem of *bandwidth allocation in a self-managing multimedia file server*. By a multimedia file server, we mean one that services a heterogeneous mix of conventional best-effort and soft real-time streaming media workloads (as opposed to continuous media servers that solely service streaming media workloads). By self-managing bandwidth allocation, we mean techniques to monitor the file server workload and react to both long-term and short-term changes in the load by dynamically allocating bandwidth to various classes. Our work has led to several research contributions.

We first identify several requirements that should be met by a dynamic bandwidth allocation technique. In particular, we argue that such a technique should (i) provide control over the time-scale of allocation and over the allocation itself, (ii) provide stable behavior during transient overloads and (iii) exploit the semantics of each application class while determining their allocations. We then present self-managing bandwidth allocation techniques for single-disk and multi-disk servers. Our techniques consist of two key components: (i) a workload monitoring module that can efficiently track the load in each application class, and (ii) a bandwidth manager that uses these workload statistics to dynamically determine the allocation of each class. A novel feature of our techniques is that they provide several tunable parameters to control the monitoring and the allocation process. We conduct an extensive simulation study of our tech-

* This research was supported in part by a NSF Career award CCR-9984030, NSF grants ANI 9977635, CDA-9502639, EIA-0080119, Intel, IBM, EMC, Sprint, and the University of Massachusetts.

ACM Multimedia '01 Ottawa, Ontario, Canada

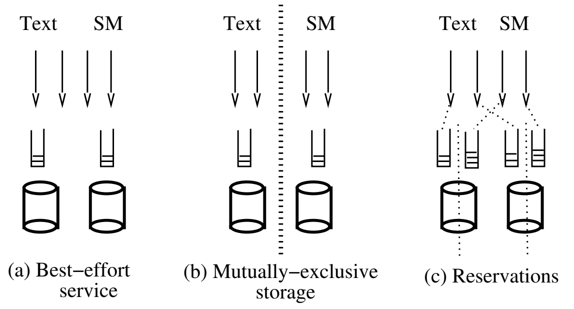


Figure 1: Three techniques for supporting multiple application classes at a file server.

niques using both synthetic and real-world trace workloads. Our experiments show that our techniques can indeed track both short-term and long-term variations in the load and allocate bandwidth to application classes accordingly. Our comparison with static allocation shows that dynamic bandwidth allocation can yield queue lengths that are 59% smaller during overloads and admit a larger number of soft real-time clients into the system.

The rest of this paper is structured as follows. In Section 2, we define the problem of self-managing bandwidth allocation in file servers. Section 3 presents a self-managing bandwidth allocation technique for single disk servers. In Section 4, we address this problem for multi-disk servers. Section 5 discusses our experimental methodology, while Section 6 presents the results of our experimental evaluation. Section 7 discusses related work, and finally, Section 8 presents our conclusions.

2. Self-Managing Bandwidth Allocation: Problem Definition

Consider a file server that services both streaming media and traditional best-effort requests. Most modern file servers belong to this category—they service requests for a mix of streaming media, image and textual data (as anecdotal evidence, consider users who store MP3 audio files and digital images along with traditional textual/numeric documents in their home directories). The workload serviced by such a file server can be broadly classified into two categories: best-effort and soft real-time. The best-effort class comprises of requests for traditional text/numeric and image data. Applications in this class need low average response times or high aggregate throughput, but do not require any performance guarantees. In contrast, the soft real-time class comprises of requests for streaming media data; applications in this class impose deadlines that must be met but can tolerate an occasional violation of these deadlines. Since the two classes have different characteristics and performance requirements, modern file servers must address the challenge of reconciling this heterogeneity.

A file server can employ one of three different techniques for managing these two classes (see Figure 1):

- *Best-effort service:* In the simplest case, the file server does not employ any specialized techniques for managing the two classes and provides a simple best-effort service to both textual and streaming media requests. In such a scenario, the performance requirements of soft real-time requests can be met only by over-engineering the capacity of the server and running the server at a low utilization levels. Since file server workloads are often bursty [11], performance guarantees of real-time requests are violated if a transient increase in the workload causes saturation. Another limitation is that requests from the two classes can interfere with one another—a burst of real-time requests can starve best-effort requests and vice versa. Due to these limitations, the overall utility of this approach to streaming media applications is often unsatisfactory.

- *Mutually exclusive storage:* An alternate approach is to store files from the two application classes on a mutually exclusive set of disks. Such a static partitioning of storage resources precludes the possibility of interference between the two classes. Moreover, guarantees of soft real-time requests can be met by employing simple admission control algorithms. Although conceptually simple, this approach has certain limitations. In particular, this approach is feasible only so long as the placement of files on disks can be carefully controlled (to ensure mutually exclusive storage of files). Unless the mapping of files to disks can be transparently handled by the file system, placing restrictions on end-users that dictate where to store each type of file is cumbersome, since users are used to the simplicity of creating and grouping arbitrary files in their directories. A more serious problem is that of performance—studies have shown that the static partitioning of storage space and disk bandwidth required by this approach results in up to a factor of six loss in performance (due to the lack of statistical multiplexing) [21].

- *Reservation-based approach:* A third approach is to share storage space among the two classes but reserve a certain fraction of the bandwidth on each disk for each class (i.e., stores files from both classes on all the disks but reserve disk bandwidth for each class). By sharing storage resources, the file server can extract statistical multiplexing gains; by reserving bandwidth, it can prevent interference among classes and meet the performance guarantees of the soft real-time class. Thus, a reservation-based approach overcomes the limitations of the previous two approaches. Let R_{rt} denote the fraction of the bandwidth reserved for the soft real-time class; the remaining fraction $R_{be} = 1 - R_{rt}$ is used (reserved) for the best-effort class. The challenge in designing a reservation-based approach lies in determining an appropriate partitioning R_{rt} and R_{be} such that both classes see acceptable performance (i.e., meet the deadlines of real-time requests while providing low average response times for best-effort requests). Modern file systems such as SGI's XFS [14] and IBM's Tiger Shark [12] support the notion of reservations. XFS, for instance, does so using its guaranteed-rate I/O feature [14].

Due to the inherent advantages and flexibility of the reservation-based approach, in the rest of this paper, we assume a file server that supports bandwidth reservations for each class.

There are several approaches for determine the aggregate bandwidth reservation for each class. In the simplest case, the partitioning of bandwidth among the two classes can be done manually. This can be done using past observations or future estimates of the load to determine the long-term usage in each class. Whereas this approach is feasible on the time-scale of days, short-term variations on the time-scale of tens of minutes or hours cannot be handled by the approach (since this would involve frequent manual intervention). Further, since the partitioning must be recomputed every so often to account for long-term variations in the load within each class, the possibility of human error can not be completely eliminated.

An alternate approach is to automate the monitoring of the workload within each class and dynamically partition the bandwidth among the two classes. We refer to such an approach as *self-managing bandwidth allocation*. By actively monitoring the load, the approach can react to workload changes on the time scale of minutes or hours. Furthermore, the approach can also handle transient overloads in the system and ensure stable overload behavior. A limitation of the approach, however, is that it increases the complexity of the file server.

As a final caveat, we distinguish between three related concepts: *bandwidth allocation, scheduling* and *admission control*. Whereas a bandwidth allocator manages resources over time-scales ranging from tens of minutes to several days, a scheduling algorithm operates over a time-scale of tens or hundreds of milliseconds. In contrast, admission control operates over the time scale of application lifetimes and is responsible for allocation of resources to individual applications. Put another way, a bandwidth allocator determines *what* fraction to allocate to each application class, the admission controller determines *how* to further partition bandwidth within each class among individual applications, and the scheduler determines *when* to service individual requests so as to enforce both class-specific and application-specific allocations.

In what follows, we first address the simpler problem of self-managing bandwidth allocation in a single disk file server and then use these insights to design a self-managing bandwidth allocator for multi-disk servers.

3. Self-Managing Bandwidth Allocation in a Single Disk Server

In this section we first present the system model assumed in our research. We then outline the requirements that must be met by a self-managing bandwidth allocator and finally present a bandwidth allocation technique that meets these requirements.

3.1 System Model

Consider a single disk file server that services two classes of applications—best-effort and soft real-time. Let us assume that the server reserves a certain fraction of the disk bandwidth for each application class. Let R_{be} and R_{tr} denote the reserved fractions, respectively, $0 \leq R_{be}, R_{rt} \leq 1$ and $R_{be} = 1 - R_{rt}$. Given the reservations R_{be} and R_{rt}, we assume that the file sever employs a disk scheduling algorithm that can enforce these allocations. A number of rate-based schedulers that support class-based bandwidth reservation have been proposed [2, 16, 17, 22, 23]. Any such scheduler is suitable for our purpose (since our bandwidth allocator does not make any specific assumptions about the scheduling algorithm). It is possible that the scheduler may itself further partition the bandwidth allocated to a class among individual applications. In this paper, we are only concerned about the aggregate bandwidth needs of each class; the partitioning of this aggregate among individual applications is an orthogonal issue (techniques for automatically allocating bandwidth to individual applications based on past usage is an interesting challenge and the subject of future research).

3.2 Requirements

Assuming the above system model, consider a bandwidth allocation technique that dynamically determines the fractions R_{be} and R_{rt} based on the load in each class. Such a self-managing bandwidth allocator should meet four key requirements.

- *Time-scale of allocation and monitoring:* Depending on the environment, bandwidth allocation can be performed on the time-scales ranging from a few minutes to tens of hours.

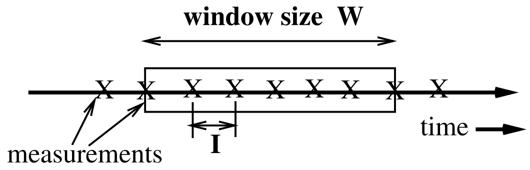


Figure 2: A Moving Histogram

  Allocating bandwidth on (small) time-scales of minutes allows the server to respond to short term variations in the load but can result in frequent fluctuations in the allocations. In contrast, allocating bandwidth on large time-scales (e.g., hours or days) allows the server to focus on long-term trends in the workload while effectively ignoring short term variations. Depending on the environment, small time-scale or large time-scale allocation or both may be necessary. A bandwidth allocator should allow a server administrator to specify the time-scale(s) of interest and recompute allocations based on this specification.

- *Control over allocations:* In addition to control over the time-scale of allocations, the bandwidth allocator should allow control over the allocation itself. Allocating bandwidth solely based on past usage can be problematic. For instance, if applications in a certain class are idle, its allocation can shrink to zero resulting in starvation for future applications. To avoid such situations, the bandwidth allocator should permit the server administrator to specify constraints on the allocations. This could be done, for instance, by specifying a set of rules that govern the actual allocations.

- *Stable overload behavior:* A bandwidth allocator should exhibit stable behavior even in the presence of transient overloads. Since the capacity of the server is exceeded during an overload, bandwidth allocation by itself can not remedy the situation. However, the allocator can (and should) make intelligent allocation decisions that prevent unstable system behavior during overloads.

- *Exploit the semantics of each class:* Requests within the best-effort class desire low average response times, while those within the real-time class have associated deadlines that must be met. Since the two classes have different performance requirements, the allocator should exploit the semantics of each class and use different criteria to allocate bandwidth to these classes. This can be achieved, for instance, by using the average load to determine the allocation of the best-effort class and the tail of the load distribution to determine the allocation of the real-time class.

Next we present our workload monitoring module and our adaptive bandwidth manager that meets these requirements.

3.3 Monitoring the Workload in the Two Classes

The workload monitoring module tracks several parameters (listed below) that are representative of the load within each class; the bandwidth manager then uses these parameters to compute the allocation of each class. For each such parameter, the monitoring module computes a probability distribution using the concept of a *moving histogram*. A moving histogram is simply a histogram computed over a moving time window. A moving histogram is

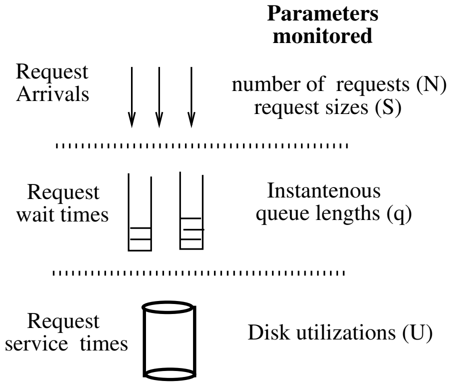


Figure 3: Parameters tracked by the monitoring module

characterized by two parameters: the window size W and the measurement interval I (see Figure 2). The window size W determines the interval of time over which the histogram is computed. Data values are recorded into the histogram every I time units. Thus, the parameter of interest is monitored over the measurement interval I and the mean value of that parameter over that interval is recorded into the histogram. The least recent value is then dropped from the histogram, effectively sliding the window by I time units. Thus, each histogram has $\lfloor \frac{W}{I} \rfloor$ data samples. By carefully choosing W and I, it is possible to exercise control over the time-scale over which the load is monitored.

The monitoring module tracks various aspects of resource usage from the time a request arrives to the time it is serviced by the disk. Monitored parameters include request arrival rates, request waiting times and disk utilizations within each class (see Figure 3):

- *Request arrival rates:* Over each interval I, the module monitors the number of request arrivals in each class (denoted by N_{be} and N_{rt}) and the request sizes (S_{be} and S_{rt}). The number of arrivals and the mean request size in that interval are then recorded into moving histograms.

- *Request waiting times:* Rather than monitoring the actual request waiting times, our monitoring module uses queue lengths as an indicator of the time each request waits in the system before it is serviced—larger the queue of outstanding requests, greater is the waiting time. This is achieved by recording the instantaneous queue lengths of the two classes (denoted by q_{be} and q_{rt}) at the end of each interval I.

- *Disk Utilizations:* The module uses the disk utilizations as a measure of the actual bandwidth consumed by each class. The utilization of a class is defined to be the fraction of the time spent by the disk in servicing requests from that class. It is computed as $U_{be} = \frac{\sum_j \tau_{be}^j}{I}$ and $U_{rt} = \frac{\sum_j \tau_{rt}^j}{I}$, where τ_{be}^j and τ_{rt}^j denote the time spent by the disk in servicing an individual best-effort and soft real-time request, respectively. The utilizations within each class are then recorded into moving histograms at the end of each interval I.

3.4 Adapting the Allocation of Each Class

The bandwidth manager uses the histograms computed by the monitoring module to periodically recompute the bandwidth allocation (reservation) of each class. The manager provides control over the time-scale of allocation using a parameter P that defines the period of these recomputations. Recall that the monitoring module uses a window size W for each moving histogram. In general, the recomputation period P can be smaller or larger than W. If allocations are recomputed more frequently than W (i.e., $P < W$) then some measurements used in the previous computations are reused to compute the new allocations (since those measurements would still be contained in the window W of the histogram). In contrast, if $P > W$, then some load measurements are never taken into account for computing the allocations. Consequently, using $P = W$ is a good rule of thumb to ensure a responsive file server. In the rest of this paper, we assume $P = W$.

The bandwidth manager uses a rule-based system to provide control over the allocation to each class. Such a rule-based system supports a set of user-defined rules that govern these allocations. Our bandwidth manager currently supports rules that specify upper and lower bounds for each class. That is, a server administrator can specify bounds (denoted by $[R_{be}^{min}, R_{be}^{max}]$ and $[R_{rt}^{min}, R_{rt}^{max}]$) on the bandwidth allocated to each class. Bounds on allocations are useful to prevent scenarios where a class receives either too little or too much bandwidth (without such bounds, the allocation of the a class could shrink to zero if the class is idle, causing starvation for newly arriving requests). We plan to support a more sophisticated set of rules as part of future work (e.g., the best-effort class should get no more than 25% of the bandwidth when the server utilization exceeds 80%).

Given the recomputation period P and bounds on the allocation of each class, the bandwidth manager estimates the bandwidth needs of each class using two metrics: (i) disk utilizations and (ii) request arrival rates.

3.4.1 Estimating Bandwidth Requirement based on Disk Utilizations

The bandwidth manager uses the moving histograms of the disk utilizations to estimate the bandwidth needs of each class. Since the two classes have different performance characteristics, a different metric is used to compute these estimates. In case of the best-effort class, the bandwidth manager uses the *median* of the utilization distribution, denoted by $Median(U_{be})$, as an estimate of the bandwidth requirement (this is because requests in this class desire low average response times).[1] In contrast, a high percentile of the utilization, denoted by $Perc(U_{rt})$, is used to estimate the requirements of the real-time class (since the tail of the distribution better reflects the needs of real-time requests). The exact percentile used to estimate the bandwidth requirements can be chosen statically or dynamically. In the latter case, the percentile could be a function of the variance in the load—the greater the variance, the higher the percentile used to estimate the bandwidth requirements. To illustrate, the percentile can be chosen as $base\_percentile + log(C_v)$ where C_v is the coefficient of variation and is computed as $C_v = \sigma(U_{rt})/E(U_{rt})$; E and σ are the mean and the standard deviation of the distribution.

After computing these utilizations, the bandwidth manager uses an exponential smoothing function to weigh the current estimate with past estimates. That is,

$$\begin{aligned} Median_\alpha(U_{be}) &= \alpha \cdot Median(U_{be}) + \\ & (1-\alpha) \cdot Median_\alpha(U_{be}) \end{aligned} \quad (1)$$

and

$$Perc_\alpha(U_{rt}) = \alpha \cdot Perc(U_{rt}) + (1-\alpha) \cdot Perc_\alpha(U_{rt}) \quad (2)$$

[1]We considered using the mean utilization for our estimate, but found the median to be a more accurate estimate due to the heavy tailed nature of the distribution.

where α is an exponential smoothing parameter, $0 \leq \alpha \leq 1$. A large value of α biases the estimates towards the immediate past measurements, whereas a small α reduces the contribution of recent measurements.

3.4.2 Estimating Bandwidth Requirement based on the Arrival Rate

Whereas the actual disk utilization is a good indicator of the needs of each class when the disk in not saturated (no overload), a different metric is needed during periods of transient overloads. This is because the total disk utilization is always 100% during an overload and no longer reflects the relative needs of each class. Consequently, the bandwidth manager uses request arrival rates to estimate the bandwidth needs of each class during transient overloads. In general, a class with larger arrival rates should be allocated a larger proportion of the disk bandwidth. Observe that since the capacity of the disk is exceeded during an overload, no allocation can actually satisfy the *total* bandwidth needs of two classes. In such a scenario, the goal of the bandwidth manager should be to ensure stable overload behavior and ensure that the allocations reflect the *relative* needs of the two classes.

To estimate the bandwidth needs based on arrival rates, the bandwidth manager first computes the number of requests arriving in each class and the request size and uses a simple disk model to estimate the bandwidth needs. As in the case of disk utilization, exponentially smoothed values of the median and a high percentile of these distributions are used for the best-effort and real-time class, respectively. Thus, the bandwidth needs of the best-effort class are computed as

$$B_{be} = Median_\alpha(N_{be}) * (t_{seek} + t_{rot} + \frac{Median_\alpha(S_{be})}{t_{xfr}}) \quad (3)$$

and those of the soft real-time class are computed as

$$B_{rt} = Perc_\alpha(N_{rt}) * (t_{seek} + t_{rot} + \frac{Perc_\alpha(S_{rt})}{t_{xfr}}) \quad (4)$$

where t_{seek}, t_{rot} and t_{xfr} denote the average seek overload, average rotational latency and the data transfer rate of the disk, respectively. Note that the first term in the above expression represents the number of disk requests, while the second term represents the time to service each disk request.

3.4.3 Computing the Reservations of Each Class

The bandwidth manager begins by initializing the allocation of each class to a user-specified value (R_{be}^{init} and R_{rt}^{init}). After each interval of P time units, the bandwidth manager estimates the bandwidth needs of each class (Section 3.4.1 and 3.4.2) and then computes the new allocations using the following algorithm.

- **Case 1:** *Neither class utilizes its entire allocation.* This scenario occurs when $Median_\alpha(U_{be}) < R_{be}$ and $Perc_\alpha(U_{rt}) < R_{rt}$. Since neither class is utilizing its entire allocation, no action is necessary. Hence, the allocations of the two classes remains unchanged.

- **Case 2:** *The best-effort class utilizes its entire allocation.* This scenario occurs when $Median_\alpha(U_{be}) \geq R_{be}$ and $Perc_\alpha(U_{rt}) < R_{rt}$. Since the best-effort class utilizes or exceeds its allocated share[2] and the real-time class is under-utilized, the bandwidth manager should increase the allocation of the best-effort class (and correspondingly decrease the allocation of the real-time class). This is achieved by setting

$$R_{be}^{new} = Median_\alpha(U_{be}) \quad (5)$$

The allocation of the real-time class is then set to $R_{rt}^{new} = 1 - R_{be}^{new}$.

- **Case 3:** *The real-time class utilizes its entire allocation.* In this scenario, $Median_\alpha(U_{be}) < R_{be}$ and $Perc_\alpha(U_{rt}) \geq R_{rt}$. Since load in the real-time class equals or exceeds its allocation, the allocation of this class should be increased appropriately. Consequently, the bandwidth manager sets the new allocation of the class to

$$R_{rt}^{new} = Perc_\alpha(U_{rt}) \quad (6)$$

The allocation of the best-effort class is set to $R_{be}^{new} = 1 - R_{rt}^{new}$.

- **Case 4:** *Overload.* An overload is said to occur when both classes use up their entire allocations (resulting in saturation) or the queue of pending requests exceeds a threshold. That is, (i) $Median_\alpha(U_{be}) \geq R_{be}$ and $Perc_\alpha(U_{rt}) \geq R_{rt}$; or (ii) $q_{be} \geq Q$ or $q_{rt} \geq Q$, where Q is a large threshold. Since disk utilizations are not representative of the relative requirements of the two classes during an overload, the bandwidth manager uses the request arrival rate to compute the allocation of each class. Given the bandwidth estimates, B_{be} and B_{rt}, based on arrival rates, the new allocations are computed as

$$R_{be}^{new} = \frac{B_{be}}{B_{be} + B_{rt}} \quad (7)$$

and

$$R_{rt}^{new} = \frac{B_{rt}}{B_{be} + B_{rt}} \quad (8)$$

As explained earlier, the use of the relative bandwidth needs of the two classes to compute allocations results in more stable overload behavior.

The above allocations are then constrained (if necessary) using user-specified bounds $[R_{be}^{min}, R_{be}^{max}]$ and $[R_{rt}^{min}, R_{rt}^{max}]$.

Our adaptive algorithm has the following salient features: (1) it provides control over the time-scale of monitoring and allocation via two tunable parameters: $P (= W)$ and α (in general, larger recomputation periods and smaller αs bias the allocator to long-term variations in the load), (2) it allows control over the allocation via a set of rules to constrain the allocation, (3) it employs techniques to provide stable overload behavior, and (4) it exploits the semantics of each class by using different metrics (median and percentiles of the distribution) to estimate bandwidth needs. Thus, the bandwidth allocator meets all of the requirements outlined in Section 3.2.

In what follows, we show how to enhance this technique to allocate bandwidth in muti-disk servers.

4. Self-Managing Bandwidth Allocation in a Multi-disk Server

Due to the sheer volume of data stored on servers, modern file servers employ multiple disks or disk arrays as their underlying storage medium. A multi-disk server can employ one of two placement techniques to store files—each file can be mapped to a single disk or the server can employ striping to interleave the storage of a file across multiple disks. In the former case, the load on each

[2]Depending on the scheduling algorithm, an application class might use more bandwidth than its reserved share. This happens when the other class is under-utilized and the scheduler reallocates unused bandwidth to needy applications in the first class.

disk is independent of the load on remaining disks, whereas in the latter case the load on disks are related to one another. It is trivial to extend our self-managing bandwidth allocation technique to multi-disk servers where each file maps onto a single disk—since the disk loads are independent, the allocator can monitor a disk and allocate bandwidth independently of other disks. A different technique is needed when files are striped across multiple disks or when it is desirable to treat multiple independent disks as a single logical storage device for purposes of bandwidth allocation.

One possible approach is to monitor the load on each disk and first compute the allocation on individual disks using the algorithm described in Section *3.4.3*. The actual allocation of each class is then set to the mean allocation over all disks in the array. Whereas such an approach results in satisfactory performance for the best-effort class, it can adversely affect the performance of the real-time class. This is because the load on various disks can be different and the use of the average load to determine the allocation of the real-time class can affect requests accessing heavily loaded disks. An alternate approach is to set the allocation of each class to that on the most heavily loaded disk in the system. However, a problem with the approach is that the load on the most heavily loaded disk can significantly differ from that on the average loaded disk and using the load on the former to govern the allocation on the latter can cause a mismatch between the allocation and the actual load (thereby defeating the purpose of bandwidth allocation). Thus, neither approach is satisfactory for allocating bandwidth on a disk array.

In what follows, we present a hybrid approach that takes into account the load on the heavily loaded disks as well as the average load to compute the allocations of the two classes. We use the same notation as that in the single disk case with an additional superscript to denote a particular disk (thus R^i_{be} denotes the allocation of the best-effort class on disk i). Based on the load parameters tracked by the monitoring module, we first compute the allocations on individual disks as follows:

$$R^i_{be} = \begin{cases} Median_\alpha(U^i_{be}) & \text{if no overload} \\ \frac{B^i_{be}}{B^i_{rt}+B^i_{be}} & \text{if disk } i \text{ is overloaded} \end{cases} \tag{9}$$

and

$$R^i_{rt} = \begin{cases} Perc_\alpha(U^i_{rt}) & \text{if no overload} \\ \frac{B^i_{rt}}{B^i_{rt}+B^i_{be}} & \text{if disk } i \text{ is overloaded} \end{cases} \tag{10}$$

The average allocation of the best-effort class across all disks is then $R^{avg}_{be} = avg(R^1_{be}, R^2_{be}, \ldots, R^D_{be})$ and the maximum allocation of the class on any disk is $R^{max}_{be} = max(R^1_{be}, R^2_{be}, \ldots, R^D_{be})$, where D denotes the number of disks in the array. The average and the maximum allocations of the real-time class across all disks can be computed similarly. The bandwidth manager then computes the allocation of each class as a linear combination of the average and the maximum load. That is,

$$R_{be} = \gamma \cdot R^{max}_{be} + (1-\gamma) \cdot R^{avg}_{be} \tag{11}$$

where the parameter γ, $0 \leq \gamma \leq 1$, determines the contribution of the average and the maximum load to the final allocation. Similarly, the allocation of the real-time class is

$$R_{rt} = \gamma \cdot R^{max}_{rt} + (1-\gamma) \cdot R^{avg}_{rt} \tag{12}$$

Finally, since the fractions R_{be} and R_{rt} may not sum to 1 (due to the skew between the average and maximum loads and the parameter γ), the final allocation is normalized as follows:

$$R^{new}_{be} = \frac{R_{be}}{R_{be}+R_{rt}}; \quad R^{new}_{rt} = \frac{R_{rt}}{R_{be}+R_{rt}} \tag{13}$$

As in the single-disk case, the new allocations are constrained (if necessary) using the user-specified upper and lower bounds. These allocations are then used on each individual disk for the next P time units.

Observe that Equations 11 and 12 are key to multi-disk bandwidth allocation—the choice of an appropriate γ helps balance the contribution of heavily loaded disks and average loaded disks to the final allocation for each class.

5. Experimental Methodology

We evaluate the efficacy of our self-managing bandwidth allocator using a simulation study. In what follows, we describe the simulation environment and the workload characteristics used in our experiments and then describe our experimental results.

5.1 Simulation Environment

We used an event-based disk simulator to evaluate our bandwidth allocation technique. Our simulator can simulate both single-disk and multi-disk servers. In either case, we assume that the server supports two application classes—best-effort and soft real-time. Requests from these classes are assumed to be serviced using the Cello disk scheduling algorithm [22]. The Cello disk scheduler supports reservations for each class and uses class-specific policies to service requests in the two classes; the SCAN policy is used to service best-effort requests, while SCAN-EDF is used to service real-time requests with deadlines. Note that, any other disk scheduler that supports class-specific reservations can be used in conjunction with our bandwidth allocator without significantly affecting our results. The file server is assumed to use one or more Seagate Elite-3 disks to store files from the two application classes.[3] The block size used for storing text files is assumed to be 4KB, while that for the video files is 64KB. In case of disk-arrays (i.e., a multi-disk server), all files are assumed to be striped across disks in the array.

The workload monitoring module employed by the simulator tracks various load parameters as described in Section 3.3. The moving histograms computed by the module are the used by the bandwidth manager to compute the allocation for each class (as described in Sections 3.4 and 4). The allocation of each class is assumed to be initialized to $R^{init}_{be} = R^{init}_{rt} = 0.5$ at the beginning of each simulation experiment.

5.2 Workload Characteristics

We use two types of workloads in our experiments: trace-driven and synthetic. Our trace workloads have been gathered from a real file-server and enable us to determine the efficacy of our methods for real-world scenarios. However, since a trace workload only represents a small subset of the operating region of a file server, we use synthetic workloads to systematically explore the state space. Next we describe the characteristics of the workloads used in our experiments.

5.2.1 Best-effort Text Clients

We used portions of a NFS trace gathered from an Auspex file server at Berkeley to generate the trace-driven text workload [9]. The characteristics of these workloads are shown in Table 1. We assumed a 64MB LRU buffer cache at the server and filtered out requests resulting in cache hits from the original trace; the remaining requests are assumed to result in disk accesses. Figure 4 illustrates the characteristics of the resulting workload. As shown in the

[3]The Seagate Elite disk has an average seek overhead of 11 ms, an average rotational latency of 5.55 ms and a data transfer rate of 4.6 MB/s.

Table 1: Characteristics of the Auspex NFS trace

Number of read/write operations	218724
Average bit rate (original)	218.64 KB/s
Average bit rate (with 64MB cache)	83.91 KB/s
Average inter-arrival (original)	9.14 ms
Average inter-arrival (with 64MB cache)	22.53 ms
Average request size	2048.22 bytes
Peak to average bit rate (1s intervals)	12.51

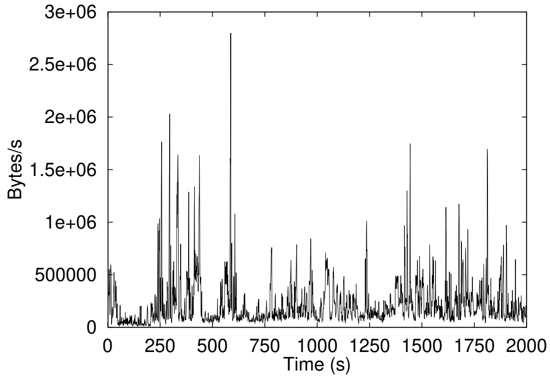


Figure 4: Bursty nature of the NFS trace workload.

figure, the text workload is very bursty; the peak to average bit rate of the trace was measured to be 12.5.

To systematically explore the state space, we also use a synthetically generated text workload. Each text client in the synthetic workload is assumed to be sequential or random. The simulator allows control over the fractions f and $1 - f$ of sequential and random text clients in the workload. Clients are assumed to arrive and depart at random time instants. Inter-arrival times of clients are assumed to be exponentially distributed. Upon arrival, each client is assumed to access a random file and file sizes (and hence, client life times) are assumed to be heavy-tailed with a Pareto distribution. These assumptions, namely exponential interarrivals and Pareto file sizes, are consistent with studies of real-world text clients [5, 11].

5.2.2 Soft Real-time Video Clients

Each video client in our simulator emulates a video player and reads a randomly selected video file at a constant frame rate (e.g., 30 frames/s). Depending on the compression algorithm, the selected video file may have a constant or a variable bit rate. Table 2 lists the characteristics of video files used in our simulations. As shown in the table, we use a mix of high bit-rate MPEG-1 files and low bit-rate MPEG-4 files. Since much of the existing online streaming media content is low bit-rate (e.g., WindowsMedia, RealMedia), this allows us to experiment with existing workloads as well future higher bit-rate workloads. All video clients are assumed to be serviced in the server-push (streaming) mode. The server services these clients in periodic rounds by retrieving a fixed number of frames in each round. Disk requests for all active video clients are issued at the beginning of each round and have the end of that round as their deadlines. The round duration was set to 1000ms in our simulations.

We used observations from a recent study of an actual streaming media workload [8] to simulate the arrival process for video clients (since the traces used in that study are not publicly available, we

Table 2: Characteristics of Video traces

File	Type	Length (frames)	Bit rate
Frasier	MPEG-1	5960	1.49 Mb/s
Newscast	MPEG-1	9000	2.33 Mb/s
Silence of the Lambs	MPEG-4	89998	107 Kb/s

couldn't use the trace itself). Video clients are assumed to arrive and depart at random instants. Inter-arrival times are exponential, the object popularity is Zipf, and the client life-times are heavy-tailed. We assumed no correlation between object sizes and object popularity, consistent with observations made in recent studies [5].

6. Experimental Evaluation

In what follows, we present the results of our experimental evaluation using the trace and synthetic workloads described in the previous section.

6.1 Ability to Adapt to Changing Workloads

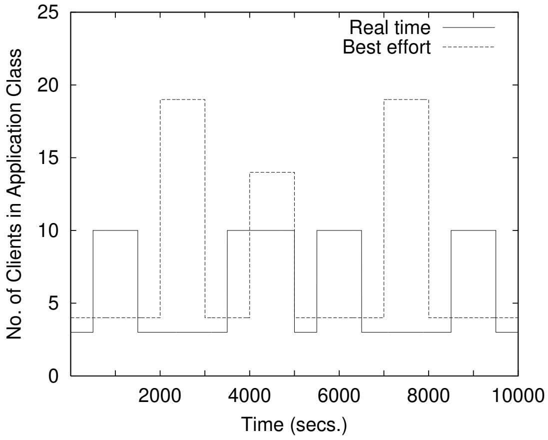
In this experiment, we show how our bandwidth allocation technique can adapt to changing workloads. We assume a single disk server and construct a workload scenario that exercises all four cases of the allocation algorithm listed in Section 3.4.3. To do so, we assume synthetic text and video clients that arrive and depart at random instants. Text clients are assumed to be sequential and access 10KB of the file every 250ms. Each video client is assumed to access a MPEG-1 file. The window size W and the recomputation period P were set to 100 seconds, the measurement interval I was 1s and the smoothing parameter α was 0.75. The percentile used for estimating the needs of the real-time class was set to 90.

Figure 5(a) depicts the variation in the number of text and video clients over the duration of the experiment (note that the figure only denotes the *number* of clients in each class, not their aggregate bandwidth requirements). We start with a small number of text and video clients at $t = 0$. At $t = 500$, there is a sudden burst of new video client arrivals (triggering case 3 in Section 3.4.3). The video burst subsides at $t = 1500$ and a burst of text clients occurs at $t = 2000$ (case 2). At $t = 4000$, there is a simultaneous burst of text and video requests, resulting in transient overload at the server (case 4).

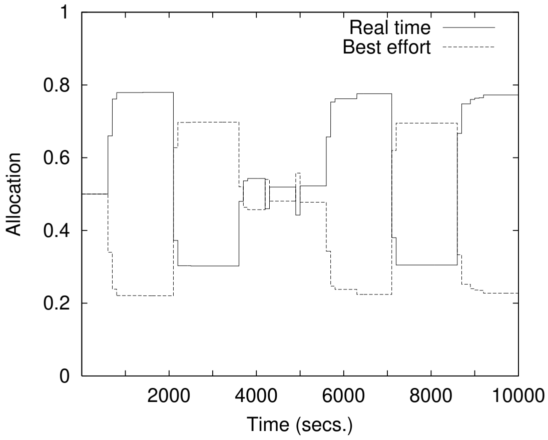
Figure 5(b) shows the allocations of the two classes for this workload, while Figures 5(c) and (d) plot the utilization of each class with the corresponding allocations. As shown in Figure 5(b), the allocation of the real-time class increases at $t = 500$ due to the video burst, while that of the best-effort class increases at $t = 2000$ due to the text burst. At $t = 4000$, the server experiences an overload and the bandwidth manager uses the request arrival rates to determine the allocations. Moreover, Figure 5(c) shows that allocation of the real-time class is always a high percentile of the load (evident from the relative values of the allocation and the utilization), whereas Figure 5(d) shows that the allocation of the best-effort class is median value of the utilization. Finally, observe that in the periods $1500 \leq t \leq 2000$ and $3000 \leq t \leq 3500$, neither class utilizes its allocated share, causing the allocations to remain unchanged (case 1).

6.2 Bandwidth Allocation in a Single-disk Server

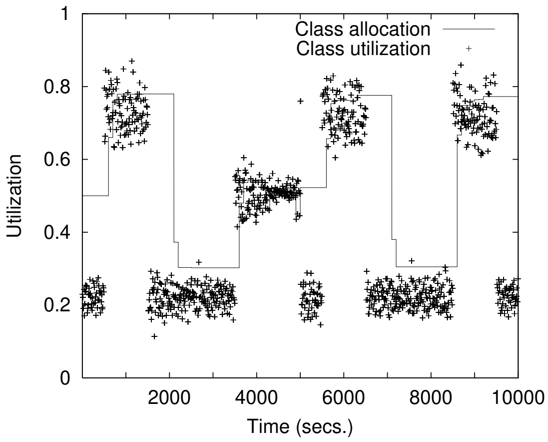
In this experiment, we demonstrate the efficacy of our approach for a single disk server. Whereas we performed experiments with both trace and synthetic workloads, due to space constraints we present our results only for trace workloads.



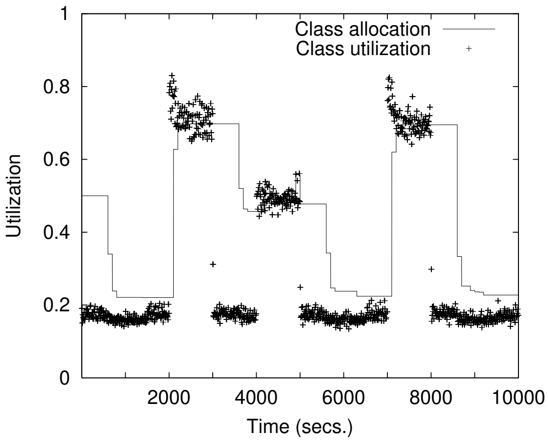
(a) Workload



(b) Allocations

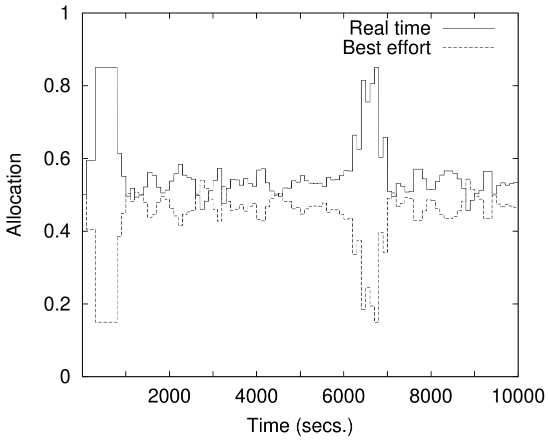


(c) Utilization of the real-time class

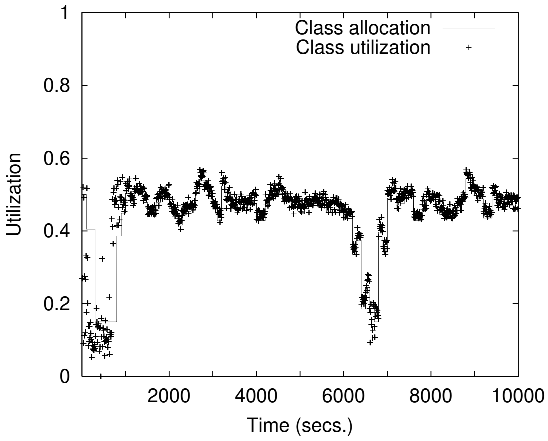


(d) Utilization of the best-effort class

Figure 5: Adaptive allocation of disk bandwidth



(a) Bandwidth allocations



(b) Utilization of the best-effort class

Figure 6: Bandwidth allocation in a single-disk server.

Our experiment uses NFS traces (with a scale factor of 3) to generate a bursty text workload[4], while keeping the video load fixed over the duration of each simulation run. We repeated the experiment for background video loads ranging from 1 to 10 simultaneous MPEG clients. This enabled us to study the impact of a bursty text load with varying background video loads. Each run simulates 2.8 hours of the workload on the file server. Note also that while the number of video clients is fixed for each simulation run, each client may impose a varying load due to the variable bit rate nature of video files.

Figures 6(a) and (b) plot the allocation of the two classes and the utilization of the best-effort class for one such combination (namely, NFS workload with 7 background video clients). As shown in the Figure 6(b), the allocation of the best-effort class closely matches the disk utilization of that class, thereby demonstrating the effectiveness of the bandwidth allocator.

6.3 Bandwidth Allocation in a Multi-disk Server

In this experiment, we demonstrate the efficacy of our approach for a multi-disk server. Like in the single disk case, we conducted experiments with both trace and synthetic workloads. Due to space constrains, we only present our results for synthetic workloads.

We assumed a multi-disk server with eight disks. Both text and video files are assumed to be striped across all disks in the array. The parameter γ that determines the contribution of the maximum load and the average load across disks was chosen to be 0.75. Like in the single disk case, we chose $W = P = 100$s and $I = 1$s.

The inter-arrival times of text clients were exponentially distributed with a mean of 10s and the lifetimes of these clients were heavy-tailed with a mean of 4 minutes. Half of the text clients were sequential and the other half random. Inter-arrival times of video clients were also exponential with a mean of 1 minute, with a heavy-tailed lifetime of 4 minutes. The popularity of video files was Zipf with a parameter of 0.47 [8]. These parameters were chosen such that the text load was mostly stable, while the video load steadily increased over the duration of the experiment, eventually resulting in an overload.

Figure 7 (a) shows the allocation of the two classes as computed by our multi-disk bandwidth allocator. Figures 7(b) and (c) plot the maximum utilization of the soft real-time class on any disk and the mean utilization across all disks, respectively (along with the corresponding allocations). As expected, we see that the allocation of the soft real-time class increases steadily with the load. Eventually, some of the disks in the array experience an overload and our allocator uses request arrival rates to compute the allocations. Note also that since we chose $\gamma = 0.75$, the allocation on the average disk is slightly larger than the utilization on that disk.

6.4 Impact of Tunable Parameters

In this section, we show how tunable parameters such as the recomputation period $P(= W)$ and the smoothing parameter α can be used to control the time-scale of bandwidth allocations.

The video load for this experiment was kept fixed over the duration of the simulation. The text load is initially steady for the first 2200 seconds and a burst occurs between $2200 \leq t \leq 2800$ (the burst is characterized by a sharp increase in the number of text clients followed by a sharp decrease). Figure 8(a) plots this variations in the text load.

We varied α from 0.25 to 1 and computed the allocations of the best-effort class. In general, a large value of α causes the bandwidth manager to maintain less history and biases the allocations towards more recent measurements. This allows the server to react to small variations in the load. In contrast, small values of α smoothes out recent variations in the load, making the server less sensitive to recent load changes. Figure 8(b) demonstrates this behavior for different values of α. As shown in the figure, when $\alpha = 1$ the bandwidth manager quickly increases the allocation of the best-effort class to match the increase in utilization due to the burst. The increase in allocation is slower for smaller values of α. For instance, when $\alpha = 0.25$ the allocation increases slowly to 60% and doesn't increase further since the burst subsides quickly.

Next, we varied P and studied its effect on the allocation. Figure 8(c) depicts the allocation of the best-effort class for different values of P. A larger recomputation period allows the bandwidth manager to focus on long-term trends and ignore short-term variations, while a smaller recomputation period enables the server to respond to short-term variations. Figure 8(c) demonstrates this behavior. When $P = 100$, the allocation of the best-effort class quickly increases to match the increase in the load. In contrast, when $P = 500$ the time-scale of interest becomes larger than the duration of the burst and consequently the bandwidth manager ignores the burst altogether and keeps the allocation unchanged.

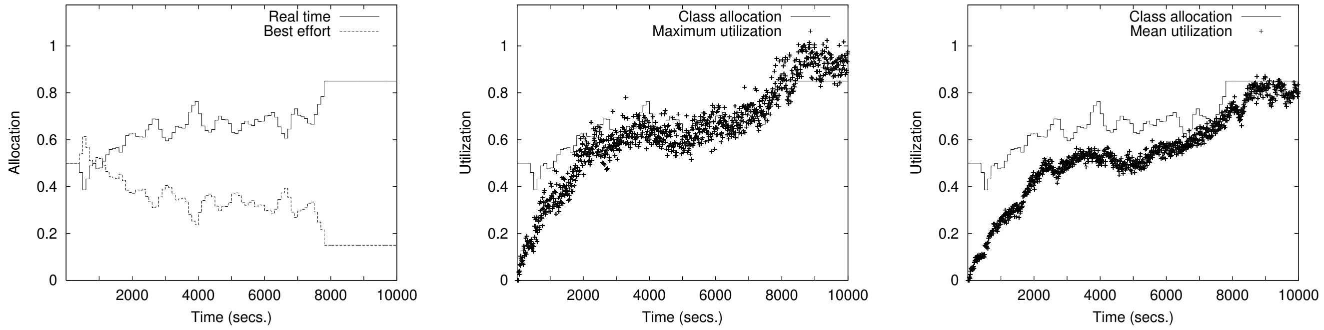
Together, these experiments demonstrate how these tunable parameters can be used to control the granularity of bandwidth allocation and the sensitiveness to load fluctuations.

6.5 Comparison with Static Allocation

Finally we demonstrate the advantages of our dynamic allocation technique over static bandwidth allocation. We initialize the allocation of the two classes to 50% of the total disk bandwidth. Whereas the allocation remains fixed for static partitioning, it varies with the load for dynamic allocation. We examine a scenario where the server experiences a transient overload due to a burst in the real-time class and measure the queue length of the real-time requests. Since the allocation remains fixed in former scenario, the server is unable to respond to an overload, causing the queue of real-time requests to grow quickly. In contrast, our bandwidth allocation technique uses request arrival rates to determine the allocation of each class and allocates a larger bandwidth to the real-time class. This enables the server to exhibit a more stable behavior during an overload, resulting in a more graceful increase in the queue length (the average queue length is also 59% smaller). We repeat the experiment with a steady video load and a burst in the best-effort class. Again, the server is unable to respond to the burst in case of static allocation,whereas our dynamic allocator allocates a larger bandwidth to the best-effort class, resulting in significantly better response times. Figures 9(a) and (b) demonstrate this behavior.

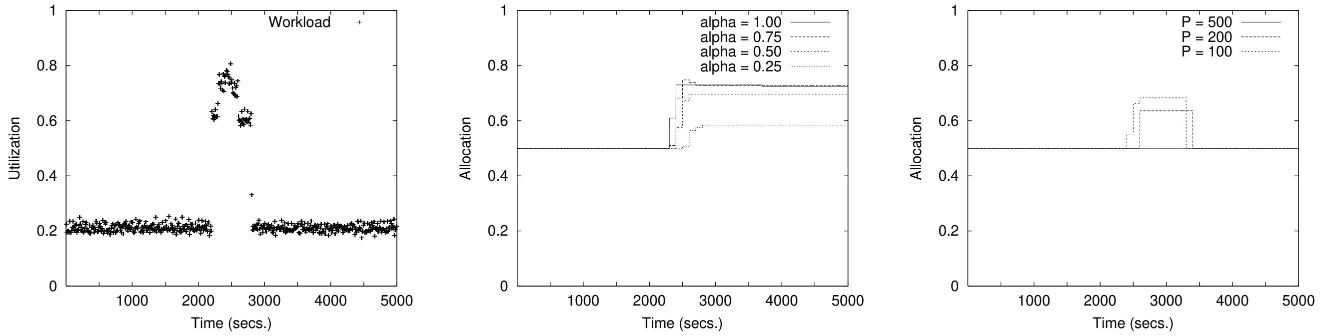
Dynamic bandwidth allocation can also be advantageous when the server employs admission control for the real-time class. If the server were to employ static bandwidth allocation, then the admission controller would only admit as many clients as the allocation of the real-time class permits; additional real-time clients would be rejected from the system even when the best-effort class is not using its entire allocation (i.e., the system has spare capacity). In contrast, dynamic bandwidth allocation allows the server to gradually increase the allocation of the real-time class based on its usage, thereby allowing the admission controller to admit additional clients. This results in more judicious use of system resources. We compared static allocation to our dynamic allocation technique in the presence of admission control in the real-time class. Our experiment consisted of a fixed text load and a video arrival every 500s. The initial allocation of the two classes was 50%. As shown in Figure 9(c), dynamic bandwidth allocation permits additional clients to be admitted into the system so long as there is unused bandwidth

[4] The scale factor scales the interarrival times of requests and allows control over the burstiness of the workload.



(a) Allocations (b) Maximum Utilization of RT class (c) Mean Utilization of RT class

Figure 7: Bandwidth allocation in a multi-disk server.



(a) Utilization (b) Effect of α (c) Effect of P

Figure 8: Effect of various tunable parameters on the granularity of bandwidth allocations.

in the best-effort class. Together these experiments demonstrate the benefits of a dynamic bandwidth allocation over static allocation.

7. Related Work

A number of recent and ongoing research efforts have focused on the design of self-managing systems [15, 19]. The IStore project, for instance, is investigating the design of work-load monitoring and adaptive resource management techniques for data-intensive network services [6]. Unlike their focus on data-intensive network applications, the focus of our work is on mixed (best-effort and streaming media) workloads. The VINO project has investigated the design of self-managing techniques for various OS tasks such as paging, interrupt latency and disk waits [20]. The design of feedback-driven proportionate allocation of disk bandwidth has also been studied [18]. This work comes closest to our current effort; the key difference is our focus on multi-disk servers, whereas they address the problem for single disk servers using control theoretic techniques. Research on storage systems at HP Labs has also investigated various issues in self managing systems such as self-configuration (Minerva [1]), capacity planning [3] and goal-based storage management [1]. Finally, a number of predictable disk scheduling algorithms have been proposed [2, 16, 17, 22, 23]. As indicated earlier, these efforts are complementary to our effort, since our bandwidth allocator can coexist with any such scheduler.

8. Concluding Remarks

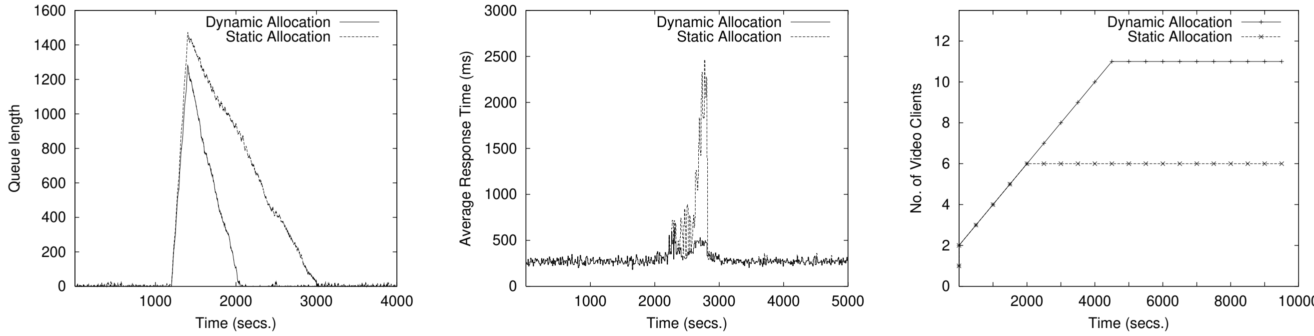
In this paper, we argued that manageability of file servers is just as important, if not more, as performance. We focused on the problem of self-managing bandwidth allocation to improve the manageability of modern file servers. We presented two techniques for dynamic bandwidth allocation—one for single disk servers and the other for servers employing multiple disks or disk arrays. Both techniques consist of two components: a workload monitoring module that efficiently monitors the load in each application class and a bandwidth manager that uses these workload statistics to dynamically determine the allocation of each class. We evaluated the efficacy of our techniques via a simulation study using synthetic and trace workloads. Our results showed that these techniques (i) provide control over the time-scale of allocation via tunable parameters, (ii) have stable behavior during overload, and (iii) provide significant advantages over static bandwidth allocation. As part of future work, we plan to develop a more sophisticated rule-based system to provide better user control over the allocations. We also plan to examine other aspects of self-managing file systems such as placement and failure handling.

9. REFERENCES

[1] G. Alvarez, K Keeton, A Merchant, E Riedel, and J Wilkes. Storage Systems Management. Tutorial presented at ACM Sigmetrics 2000, Santa Clara, CA, June 2000.

[2] P. Barham. A Fresh Approach to File System Quality of Service. In *Proceedings of NOSSDAV'97, St. Louis, Missouri*, pages 119–128, May 1997.

[3] E. Borowsky, R. Golding, P. Jacobson, A. Merchant, L. Schreier, M. Spasojevic, and J. Wilkes. Capacity Planning with Phased Workloads. In *Proceedings of WOSP'98, Santa*



(a) Queue lengths of the real-time class (b) Text response times (c) Impact of admission control

Figure 9: Comparison with Static Partitioning

Fe, NM, October 1998.

[4] E. Borowsky, R. Golding, A. Merchant, E. Shriver, M. Spasojevic, and J. Wilkes. Eliminating Storage Headaches Through Self-Management. In *Proc. of the First Symposium on Operating System Design and Implementation (OSDI), Seattle, WA*, October 1996.

[5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of Infocom'99, New York, NY*, March 1999.

[6] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D.A. Patterson. ISTORE: Introspective Storage for Data-Intensive Network Services. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII), Rio Rico, AZ*, March 1999.

[7] A. Brown and D. A. Patterson. Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the USENIX Annual Technical Conference, San Diego, CA*, June 2000.

[8] M. Chesire, A Wolman, G. Voelker, and H. Levy. Measurement and Analysis of a Streaming Workload. In *Proceedings of the USENIX Symposium on Internet Technology and Systems (USEITS), San Francisco, CA*, March 2001.

[9] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *Proceedings of ACM SIGMETRICS'94*, May 1994.

[10] J. Gray and P. Shenoy. Rules of Thumb in Data Engineering. In *Proceedings of the 16th IEEE International Conference on Data Engineering (ICDE), San Diego, CA*, pages 3–10, March 2000.

[11] S. D. Gribble, G. Manku, D. Roselli, E. Brewer, T. Gibson, and E. Miller. Self-Similarity in File Systems. In *Proceedings of ACM SIGMETRICS '98, Madison, WI*, June 1998.

[12] R. Haskin. Tiger Shark–A Scalable File System for Multimedia. *IBM Journal of Research and Development*, 42(2):185–197, March 1998.

[13] J. Hennessy. The Future of Systems Research. *IEEE Computer*, pages 27–33, August 1999.

[14] M. Holton and R. Das. XFS: A Next Generation Journalled 64-bit File System with Guaranteed Rate I/O. Technical report, Silicon Graphics, Inc, Available online as http://www.sgi.com/Technology/xfs-whitepaper.html, 1996.

[15] K. Keeton, D A. Patterson, and J. Hellerstein. The Case for Intelligent Disks (IDisks). In *Proceedings of the 24th Conference on Very LArge Databases (VLDB)*, August 1998.

[16] A. Molano, K. Juvva, and R. Rajkumar. Real-time File Systems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. In *Proceedings of IEEE Real-time Systems Symposium*, December 1997.

[17] G. Nerjes, P. Muth, M. Paterakis, Y. Romboyannakis, P. Triantafillou, and G. Weikum. Scheduling Strategies for Mixed Workloads in Multimedia Information Servers. In *Proceedings of the 8th International Workshop on Research Issues in Data Engineering (RIDE'98), Orlando, Florida*, February 1998.

[18] D. Revel, D. McNamee, C. Pu, D. Steere, and J. Walpole. Feedback Based Dynamic Proportion Allocation for Disk I/O. Technical Report CSE-99-001, OGI CSE, January 1999.

[19] E. Riedel, G A. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *Proceedings of the 24th international Conference on Very Large Databases (VLDB '98), New York, NY*, August 1998.

[20] M. Seltzer and C. Small. Self-Monitoring and Self-Adapting Systems. In *Proceedings of the 1997 Workshop on Hot Topics on Operating Systems, Chatham, MA*, May 1997.

[21] P. Shenoy, P. Goyal, and H M. Vin. Architectural Considerations for Next Generation File Systems. In *Proceedings of the Seventh ACM Multimedia Conference, Orlando, FL*, November 1999.

[22] P Shenoy and H M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In *Proceedings of ACM SIGMETRICS Conference, Madison, WI*, pages 44–55, June 1998.

[23] R. Wijayaratne and A. L. N. Reddy. Providing QoS Guarantees for Disk I/O. Technical Report TAMU-ECE97-02, Department of Electrical Engineering, Texas A&M University, 1997.