A Practical Learning-Based Approach for Dynamic Storage Bandwidth Allocation*

Vijay Sundaram and Prashant Shenoy

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

Abstract. In this paper, we address the problem of dynamic allocation of storage bandwidth to application classes so as to meet their response time requirements. We present an approach based on reinforcement learning to address this problem. We argue that a simple learning-based approach may not be practical since it incurs significant memory and search space overheads. To address this issue, we use application-specific knowledge to design an efficient, practical learning-based technique for dynamic storage bandwidth allocation. Our approach can react to dynamically changing workloads, provide isolation to application classes and is stable under overload. We implement our techniques into the Linux kernel and evaluate it using prototype experimentation and trace-driven simulations. Our results show that (i) the use of learning enables the storage system to reduce the number of QoS violations by a factor of 2.1 and (ii) the implementation overheads of employing such techniques in operating system kernels is small.

1 Introduction

Enterprise-scale storage systems may contain tens or hundreds of storage devices. Due the sheer size of these systems and the complexity of the application workloads that access them, storage systems are becoming increasingly difficult to design, configure, and manage. Traditionally, storage management tasks have been performed manually by administrators who use a combination of experience, rules of thumb, and in some cases, trial and error methods. Numerous studies have shown that management costs far outstrip equipment costs and have become the dominant fraction of the total cost of ownership of large computing systems [15]. These arguments motivate the need to automate simple storage management tasks so as to make the system self-managing and reduce the total cost of ownership.

In this paper, we address the problem of automating the task of storage bandwidth allocation to applications. We assume that the storage system is accessed by applications that can be categorized into different classes; each class is assumed to impose a certain QoS requirement. The workload seen by an application class varies over time, and we address the problem of how to allocate storage bandwidth to classes in presence of varying workloads so that their QoS needs are met. Since data accessed by applications may be stored on overlapping set of storage devices, the system must dynamically partition the device bandwidth among classes to meet their needs.

* This research was supported in part by NSF grants CCR-9984030, ANI-9977635 and EIA-0080119.

Our work on dynamic storage bandwidth allocation has led to several contributions. First, we identify several requirements that should be met by a dynamic allocation technique. We argue that such a technique (i) should adapt to varying workloads, (ii) should not violate the performance requirement of one class to service another class better, and (iii) should exhibit stable behavior under transient or sustained overloads.

Second, we design a dynamic bandwidth allocation technique based on *reinforcement learning* to meet these requirements. The key idea in such an approach is to learn from the impact of past actions and use this information to make future decisions. This is achieved by associating a cost with each action and using past observations to take an action with the least cost. We show that a simple learning approach that systematically searches through all possible allocations to determine the "correct" allocation for a particular system state has prohibitive memory and search space overheads for practical systems. We design an enhanced learning-based approach that uses domain-specific knowledge to substantially reduce this overhead (for example, by eliminating searching through allocations that are clearly incorrect for a particular system state). A key advantage of using reinforcement learning is that no prior training of the system is required; our technique allows the system to learn online.

Third, we implement our techniques into the Linux kernel and evaluate it using prototype experimentation and simulation of synthetic and trace-driven workloads. Our results show that (i) the use of learning enables the storage system to reduce the number of QoS violations by a factor of 2.1 and (ii) the implementation overheads of employing such techniques in operating system kernels is small. Overall, our work demonstrates the feasibility of using reinforcement learning techniques to automate storage bandwidth allocation in practical systems. Moreover, our techniques are sufficiently general and can be used to manage other system resources as well.

The rest of the paper is structured as follows. In Section 2, we define the problem of dynamic storage bandwidth allocation. Section 3 presents a learning-based approach for dynamic bandwidth allocation. Section 4 presents details of our prototype implementation in Linux. Section 5 presents the results of our experimental evaluation. Section 6 discusses related work, and finally, Section 7 presents our conclusions.

2 Dynamic Storage Bandwidth Allocation: Problem Definition

2.1 Background and System Model

An enterprise storage system consists of a large number of disks that are organized into disk arrays. A disk array is a collection of physical disks that presents an abstraction of a single large logical storage device; we refer to this abstraction as a logical unit (LU). An application, such as a database or a file system, is allocated storage space by concatenating space from one or more LUs; the concatenated storage space is referred to as a logical volume (LV). Figure 1 illustrates the mapping from LVs to LUs.

We assume that the workload accessing each logical volume can be partitioned into *application classes*. This grouping can be determined based on either the files accessed or the QoS requirements of requests. Each application class is assumed to have a certain response time requirement. Application classes compete for storage bandwidth and the bandwidth allocated to a class governs the response time of its requests.

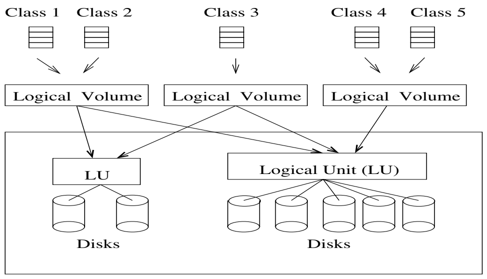


Fig. 1. Relationship between application classes, logical volumes and logical units.

To enable such allocations, each disk in the system is assumed to employ a QoS-aware disk scheduler (such as [7,18,23]). Such a scheduler allows disk bandwidth to be reserved for each class and enforces these allocations at a fine time scale. Thus, if a certain disk receives requests from n application classes, then we assume that the system dynamically determines the reservations $R_1, R_2, \cdots R_n$ for these classes such that the response time needs of each class are met and $\sum_{i=1}^{n} R_i = 1$ (the reservation R_i essentially denotes the fraction of the total bandwidth allocated to class i; $0 \leq R_i \leq 1$).

2.2 Key Requirements

Assuming the above system model, consider a bandwidth allocation technique that dynamically determines the reservations $R_1, R_2, ..., R_n$ based on the requirements of each class. Such a scheme should satisfy the following key requirements.

Meet class response time requirements: Assuming that each class specifies a target response-time d_i, the bandwidth allocation techniques should allocate sufficient bandwidth to each class to meet its target response-time requirements. Whether this goal can be met depends on the load imposed by each application class and the aggregate load. In scenarios where the response time needs of a class can not be met (possibly due to overload), the bandwidth allocation technique should attempt to minimize the difference between the observed and the target response times.

Performance isolation: Whereas the dynamic allocation technique should react to changing workloads, for example, by allocating additional bandwidth to classes that see an increased load, such increases in allocations should not affect the performance of less loaded classes. Thus, only spare bandwidth from underloaded classes should be reallocated to classes that are heavily loaded, thereby isolating underloaded classes from the effects of overload.

Stable overload behavior: Overload is observed when the aggregate workload exceeds disk capacity, causing the target response times of all classes to be exceeded. The bandwidth allocation technique should exhibit stable behavior under overload. This is especially important for a learning-based approach, since such techniques systematically search though various allocations to determine the correct allocation; doing so under overloads can result in oscillations and erratic behavior. A well-designed dynamic allocation scheme should prevent such unstable system behavior.

2.3 Problem Formulation

To precisely formulate the problem addressed in this paper, consider an individual disk from a large storage system that services requests from n application classes. Let $d_1, d_2, \ldots, d_n$ denote the target response times of these classes. Let $Rt_1, Rt_2, \ldots, Rt_n$ denote the response time of these classes observed over a period P. Then the dynamic allocation technique should compute reservations $R_1, R_2, \cdots, R_n$ such that $Rt_i \leq d_i$ for any class i subject to the constraint $\sum_i R_i = 1$ and $0 \leq R_i \leq 1$. Since it may not always be possible to meet the response time needs of each class, especially under overload, we modify the above condition as follows: instead of requiring $Rt_i \leq d_i, \forall i$, we require that the response time should be less than or as close to the target as possible. That is, $(Rt_i - d_i)^+$ should be equal to or as close to zero as possible (the notation x^+ equals x for positive values of x and equals 0 for negative values). Instead of attempting to meet this condition for each class, we define a new metric

$$sigma_{rt}^{+} = \sum_{i=1}^{n} (Rt_i - d_i)^+ \tag{1}$$

and require that $sigma_{rt}^{+}$ be minimized. Observe that, $sigma_{rt}^{+}$ represents the aggregate amount by which the response time targets of classes are exceeded. Minimizing a single metric $sigma_{rt}^{+}$ enables the system to collectively minimize the QoS violations across application classes.

We now present a learning-based approach that tries to minimize the $sigma_{rt}^{+}$ observed at each disk while meeting the key requirements outlined in Section 2.2.

3 A Learning-Based Approach

In this section, we first present some background on reinforcement learning and then present a simple learning-based approach for dynamic storage bandwidth allocation. We discuss limitations of this approach and present an enhanced learning-based approach that overcomes these limitations.

3.1 Reinforcement Learning Background

Any learning-based approach essentially involves learning from past history. Reinforcement learning involves learning how to map situations to *actions* so as to maximize a numerical *reward* (equivalent of a *cost* or *utility* function) [21]. It is assumed that the system does not know which actions to take in order to maximize the reward; instead the system must discover ("learn") the correct action by systematically trying various actions. An *action* is defined to be one of the possible ways to react to the current system state. The system state is defined to be a subset of what can be perceived from the environment at any given time.

In the dynamic storage bandwidth allocation problem, an action is equivalent to setting the allocations (i.e., the reservations) of each class. The system state is the vector of the observed response times of the application classes. The objective of reinforcement learning is to maximize the reward despite *uncertainty* about the environment (in our

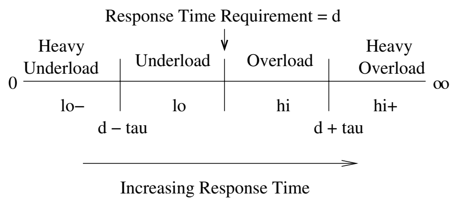


Fig. 2. Discretizing the State Space

case, the uncertainty arises due to the variations in the workload). An important aspect of reinforcement learning is that, unlike some learning approaches, no prior training of the system is necessary—all the learning occurs online, allowing the system to deal with unanticipated uncertainties (e.g., events, such as flash crowds, that can not have been anticipated in advance). It is this feature of reinforcement learning that makes it particularly attractive for our problem.

A reward function defines the goal in the reinforcement learning; by mapping an action to a reward, it determines the intrinsic desirability of that state. For the storage allocation problem, we define the reward function to be $-sigma_{rt}^{+}$—maximizing reward implies minimizing $sigma_{rt}^{+}$ and the QoS violations of classes. In reinforcement learning, we use reward values learned from past actions to estimate the expected reward of a (future) action.

With the above background, we present a reinforcement learning approach based on *action values* to dynamically allocate storage bandwidth to classes.

3.2 System State

A simple definition of system state is a vector of the response times of the n classes: $(Rt_1, Rt_2, \ldots, Rt_n)$, where Rt_i denotes the mean response time of class i observed over a period P. Since the response time of a class can take any arbitrary value, the system state space is theoretically infinite. Further, the system state by itself does not reveal if a particular class has met its target response time. Both limitations can be addressed by discretizing the state space as follows: partition the range of the response time (which is $[0, \infty)$) into four parts

$$\{[0, d_i - \tau_i], (d_i - \tau_i, d_i], (d_i, d_i + \tau_i], (d_i + \tau_i, \infty)\}$$

and map the observed response time Rt_i into one of these sub-ranges (τ_i is a constant). The first range indicates that the class response time is substantially below its target response time (by a threshold τ_i). The second (third) range indicates that the response time is slightly below (above) the target and by no more than the threshold τ_i. The fourth range indicates a scenario where the target response time is substantially exceeded. We label these four states as lo^{-}, lo, hi and hi^{+}, respectively, with the labels indicating different degrees of over- and under-provisioning of bandwidth (see Figure 2). The state of a class is defined as $S_i \in \{lo^{-}, lo, hi, hi^{+}\}$ and the modified state space is a vector of these states for each class: $S = (S_1, S_2, \ldots, S_n)$. Observe that, since state of a class can take only four values, the potentially infinite state space is reduced to a size of 4^n.

3.3 Allocation Space

The reservation of a class R_i is a real number between 0 and 1. Hence, the allocation space $(R_1, R_2, \ldots, R_n)$ is infinite due to the infinitely many allocations for each class. Since a learning approach must search through all possible allocations to determine an appropriate allocation for a particular state, this makes the problem intractable. To discretize the allocation space, we impose a restriction that requires the reservation of a class be modified in steps of T, where T is an integer. For instance, if the step size is chosen to be 1% or 5%, the reservation of a class can only be increased or decreased by a multiple of the step size. Imposing this simple restriction results in a finite allocation space, since the reservation of a class can only take one of m possible values, where $m = 100/T$. With n classes, the number of possible combinations of allocations is $\binom{m+n-1}{m}$, resulting in a finite allocation space. Choosing an appropriate step size allows allocations to be modified at a sufficiently fine grain, while keeping the allocation space finite. In the rest of this paper, we use the terms *action* and *allocation* interchangeably.

3.4 Cost and State Action Values

For the above definition of state space, we observe that the response time needs of a class are met so long it is in the lo^- or lo states. In the event an application class is in hi or hi^+ states, the system needs to increase the reservations of the class, assuming spare bandwidth is available, to induce a transition back to lo^- or lo. This is achieved by computing a new set of reservations $(R_1, R_2, \ldots, R_n)$ so as to maximize the reward $-sigma^+_{rt}$. Note that the maximum value of the reward is zero, which occurs when the response time needs of all classes are met (see Equation 1).

A simple method for determining the new allocation is to pick one based on the observed rewards of previous actions from this state. An action (allocation) that resulted in largest reward $(-sigma^+_{rt})$ is likely to do so again and is chosen over other lower reward actions. Making this decision requires that the system first try out all possible actions, possibly multiple times, and then choose one that yields the largest reward. Over a period of time, each action may be chosen multiple times and we store an exponential average of the observed reward from this action (to guide future decisions):

$$Q^{new}_{(S_1,S_2,\ldots,S_n)}(a) = \gamma * Q^{old}_{(S_1,S_2,\ldots,S_n)}(a) + (1-\gamma) * -sigma^+_{rt}(a) \qquad (2)$$

where Q denotes the exponentially averaged value of the reward for action a taken from state $(S_1, S_2, \ldots, S_n)$ and γ is the exponential smoothing parameter (also known as the *forgetting* factor). Learning methods of this form, where the actions selected are based on estimates of action-reward values (also referred to as action values), are referred to as *action-value methods*.

We choose an exponential average over a sample average because the latter is appropriate only for stationary environments. In our case, the environment is non-stationary due to the changing workloads and the same action from a state may yield different rewards depending on the current workload. For such scenarios, recency-weighted exponential averages are more appropriate. With 4^n states and $\binom{m+n-1}{m}$ possible actions in each state, the system will need to store $\binom{m+n-1}{m} * 4^n$ such averages, one for each action.

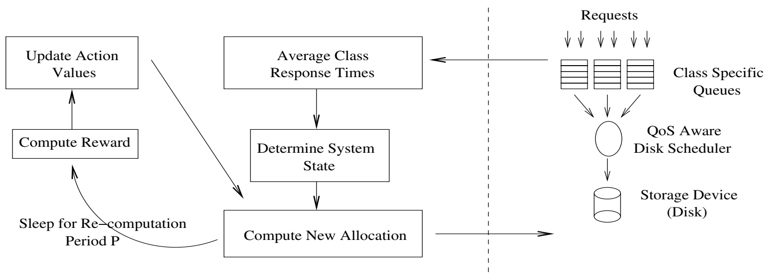


Fig. 3. Steps involved in learning.

3.5 A Simple Learning-Based Approach

A simple learning approach is one that systematically tries out all possible allocations from each system state, computes the reward for each action and stores these values to guide future allocations. Note that it is the discretization of the state space and the allocation space as described in Sections 3.4 and 3.2 which make this approach possible. Once the reward values are determined for the various actions, upon a subsequent transition to this state, the system can use these values to pick an allocation with the maximum reward. The set of learned reward values for a state is also referred to as the *history* of the state. As an example, consider two application classes that are allocated 50% each of the disk bandwidth and are in (lo^-, lo^-). Assume that a workload change causes a transition to (lo^-, hi^+). Then the system needs to choose one of several possible allocations: $(0, 100)$, $(5, 95)$, $(10, 90)$, ..., $(100, 0)$. Choosing one of these allocations allows the system to learn the reward $-sigma_{rt}^+$ that accrues as a result of that action. After trying all possible allocations, the system can use these learned values to directly determine an allocation that maximizes reward (by minimizing the aggregate QoS violations). This quicker and suitable reassignment of class allocations is facilitated by learning. Figure 3 shows the steps involved in a learning based approach.

Although such a reinforcement learning scheme is simple to design and implement, it has numerous drawbacks.

Actions are oblivious of system state: A key drawback of this simple learning approach is that the actions are oblivious of the system state—the approach tries all possible actions, even ones that are clearly unsuitable for a particular state. In the above example, for instance, any allocation that decreases the share of the overloaded hi^+ class and increases that of the underloaded lo^- class is incorrect. Such an action can worsen the overall system performance. Nevertheless, such actions are explored to determine their reward. The drawback arises primarily because the semantics of the problem are not incorporated into the learning technique.

No performance isolation: Since the system state is not taken into account while making allocation decisions, the approach can not provide performance isolation to classes. In the above example, an arbitrary allocation of $(0, 100)$ can severely affect the lo^- class while favoring the overloaded class.

Large search space and memory requirements: Since there are $\binom{m+n-1}{m}$ possible allocations in each of the 4^n states, a systematic search of all possible allocations is impractical. This overhead is manageable when $n = 2$ classes and $m = 20$ (which corresponds to a step size of 5%; $m = 100/5$), since there are only $\binom{21}{20} = 21$ allocations for each of the $4^2 = 16$ states. However, for $n = 5$ classes, the number of possible actions increases to 10626 for each of the 4^5 states. Since the number of possible actions increases exponentially with increase in the number of classes, so does the memory requirement (since the reward for each allocation needs to be stored in memory to guide future allocations). For $n = 5$ classes and $m = 20$, 83MB of memory is needed per disk to store these reward values. This overhead is impractical for storage systems with large number of disks.

3.6 An Enhanced Learning-Based Approach

In this section, we design an enhanced learning approach that uses the semantics of the problem to overcome the drawback of the naive learning approach outlined in the previous section. The key insight used in the enhanced approach is to use the state of a class to determine whether to increase or decrease its allocation (instead of naively exploring all possible allocations). In the example listed in the previous section, for instance, only those allocations that increase the reservation of the overloaded class and decrease the allocation of the underloaded class are considered. The technique also includes provisions to provide performance isolation, achieve stable overload behavior, and reduce memory and search space overheads.

Initially, we assume that the allocations of all classes are set to a default value (a simple default allocation is to assign equal shares to the classes; any other default may be specified). We assume that the allocations of classes are recomputed every P time units. To do so, the technique first determines the system state and then computes the new allocation for this state as follows:

Case I: *All classes are underloaded (are in lo^- or lo).* Since all classes are in lo or lo^-, by definition, their response time needs are satisfied and no action is necessary. Hence, the allocation is left unchanged. An optimization is possible when some classes are in lo^- and some are in lo. Since the goal is to drive all classes to as low as state as possible, one can reallocate bandwidth from the classes in lo^- to the classes in lo. How bandwidth is reallocated and history maintained to achieve this is similar to the approach described in Case III below.

Case II: *All classes are overloaded (are in hi or hi^+).* Since all classes are in hi or hi^+, the target response times of all classes are exceeded, indicating an overload situation. While every class can use extra bandwidth, none exists in the system. Since no spare bandwidth is available, we leave the allocations unchanged.

An additional optimization is possible in this state. If some class is heavily overloaded (i.e., is in hi^+) and is currently allocated less than its initial default allocation, then the allocation of all classes is set to their default values (the allocation is left unchanged otherwise). The insight behind this action is that no class should be in hi^+ due to starvation resulting from an allocation less than its default. Resetting the allocations to their default values during such heavy overloads ensures that the system performance is no worse than a static approach that allocates the default allocation to each class.

Case III: *Some classes are overloaded, others are underloaded (some in hi^+ or hi and some in lo or lo^-).* This is the scenario where learning is employed. Since some classes are underloaded while others are overloaded, the system should reallocate spare band-

width from underloaded classes to overloaded classes. Initially, there is no history in the system and the system must *learn* how much bandwidth to reassign from underloaded to overloaded classes. Once some history is available, the reward values from past actions can be used to guide the reallocation.
The learning occurs as follows. The application classes are partitioned into two sets: *lenders* and *borrowers*. A class is assigned to the lenders set if it is in lo or lo^-; classes in hi and hi^+ are deemed borrowers. The basic idea is to reduce the allocation of a lender by T and reassign this bandwidth to a borrower. Note that the bandwidth of only one lender and one borrower is modified at any given time and only by the step size T; doing so systematically reassigns spare bandwidth from lenders to borrowers, while learning the rewards from these actions.
Different strategies can be used to pick a lender and a borrower. One approach is to pick the most needy borrower and the most over-provisioned lender (these classes can be identified by how far the class is from its target response time; the greater this difference, the greater the need or the available spare bandwidth). Another approach is to cycle through the list of lenders and borrowers and reallocate bandwidth to classes in a round-robin fashion. The latter strategy ensures that the needs of all borrowers are met in a cyclic fashion, while the former strategy focuses on the most needy borrower before addressing the needs of the remaining borrowers.
Regardless of the strategy, the system state is recomputed P time units after each reallocation. If some classes continue to be overloaded, while others are underloaded, we repeat the above process. If the system transitions to a state defined by Case I or II, we handle them as discussed above.
The reward obtained after each allocation is stored as an exponentially-smoothed average (as shown in Equation 2). However, instead of storing the rewards of all possible actions, we only store the rewards of the actions that yield the k highest rewards. The insight here is that the remaining actions do not yield a good reward and, since the system will not consider them subsequently, we do not need to store the corresponding reward values. These actions and their corresponding reward estimates are stored as a link list, with the neighboring elements in the link list differing in the allocations of two classes by the step size T, that of one lender and one borrower. This facilitates a systematic search of the suitable allocation for a state, and also pruning of the link list to maintain a size of no more than k. By storing a fixed number of actions and rewards for any given state, the memory requirements can be reduced substantially. Further, while the allocation of a borrower and a lender is changed only by T in each step during the initial learning process, these can be changed by a larger amount subsequently once some history is available (this is done by directly picking the allocation that yields the maximum reward).

As a final optimization, we use a small non-zero probability ϵ to bias the system to occasionally choose a neighboring allocation instead of the allocation with the highest reward (a neighboring allocation is one that differs from the best allocation by the step size T for the borrowing and lending classes, e.g., $(30, 70)$ instead of $(35, 65)$ when $T = 5\%$). The reason we do this is that it is possible the value of an allocation is underestimated as a result of a sudden workload reversal, and the system may thus select the best allocation based on the current history. An occasional choice of a neighboring allocation ensures that the system explores the state space sufficiently well to discover a suitable allocation.

Observe that our enhanced learning approach reclaims bandwidth only from those classes that have bandwidth to spare (lo and lo^- classes) and reassigns this bandwidth to classes that need it. Since a borrower takes up bandwidth in increments of T from a lender, the lender could in the worst case end up in state hi[1]. At this stage there would be a state change, and the action would be dictated by this new state. Thus, this strategy ensures that any new allocation chosen by the approach can only improve (and not worsen) the system performance; doing so also provides a degree of performance isolation to classes.

The technique also takes the current system state into account while making allocation decisions and thereby avoids allocations that are clearly inappropriate for a particular state; in other words, the optimized learning technique intelligently guides and restricts the allocation space explored. Further, since only the k highest reward actions are stored, the worst case search overhead is reduced to $O(k)$. This results in a substantial reduction from the search overheads of the simple learning approach. Finally, the memory needs of the technique reduce from $\binom{m+n-1}{m}$ to $4^n * k$, where k is the number of high reward actions for which history is maintained. This design decision also results in a substantial reduction in the memory requirements of the approach. In the case of 5 application classes, $T = 5\%$ (recall $m = 100/T$) and $k = 5$, for example, the technique yields more than 99% reduction in memory needs over the simple learning approach.

4 Implementation in Linux

We have implemented our techniques in the Linux kernel version 2.4.9. Our prototype consists of three components: (i) a QoS-aware disk scheduler that supports per-class reservations, (ii) a module that monitors the response time requirements of each class, and (iii) a learning-based bandwidth allocator that periodically recomputes the reservations of the classes on each disk. Our prototype was implemented on a Dell PowerEdge server (model 2650) with two 1 GHz Pentium III processors and 1 GB memory that runs RedHat Linux 7.2. The server was connected to a Dell PowerVault storage pack (model 210) with eight SCSI disks. Each disk is a 18GB 10,000 RPM Fujitsu MAJ3182MC disk[2]. We use the software RAID driver in Linux to configure the system as a single RAID-0 array.

We implement the Cello QoS-aware disk scheduler in the Linux kernel [18]. The disk scheduler supports a configurable number of application classes and allows a fraction of the disk bandwidth to be reserved for each class (these can be set using the scheduler system call interface). These reservations are then enforced on a fine time scale, while taking disk seek overheads into account. We extend the *open* system call to allow applications to associate file I/O with an application class; all subsequent read and write operations on the file are then associated with the specified class. The use of our enhanced open

[1] The choice of the step size T is of importance here. If the step-size is too big the overloaded class could end up in underload and vice versa and this could result in oscillations.

[2] The Fujitsu MAJ3182MC disk has an average seek overhead of 4.7 ms, an average latency of 2.99 ms and a data transfer rate of 39.16 MB/s.

system call interface requires application source code to be modified. To enable legacy application to benefit from our techniques, we also provide a command line utility that allows a process (or a thread) to be associated with an application class—all subsequent I/O from the process is then associated with that class. Any child processes that are forked by this process inherit these attributes and their I/O requests are treated accordingly.

We also add functionality into the Linux kernel to monitor the response times of requests in each class (at each disk); the response time is defined to the sum of the queuing delay and the disk service times. We compute the mean response time in each class over a moving window of duration P.

The bandwidth allocator runs as a privileged daemon in user space. It periodically queries the monitoring module for the response time of each class; this can done using a special-purpose system call or via the $/proc$ interface in Linux. The response time values are then used to compute the system state. The new allocation is then determined and conveyed to the disk scheduler using the scheduler interface.

5 Experimental Evaluation

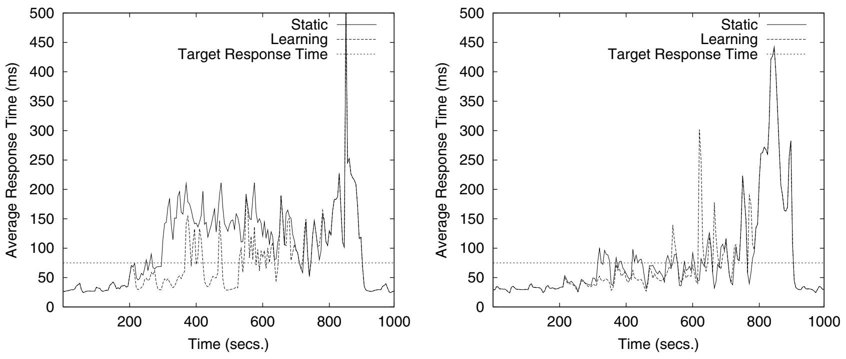
In this section, we demonstrate the efficacy of our techniques using a combination of prototype experimentation and simulations. In what follows, we first present our simulation methodology and simulation results, followed by results from our prototype implementation.

5.1 Simulation Methodology and Workload

We use an event-based storage system simulator to evaluate our bandwidth allocation technique. The simulator simulates a disk array that is accessed by multiple application classes. Each disk in the array is modeled as a 18GB 10,000 RPM Fujitsu MAJ3182MC disk. The disk array is assumed to be configured as a RAID-0 array with multiple volumes; unless specified otherwise we assume an array of 8 disks . Each disk in the system is assumed to employ a QoS-aware disk scheduler that supports class-specific reservations; we use the Cello disk scheduler [18] for this purpose. Observe that the hardware configuration assumed in our simulations is identical to that in our prototype implementation. We assume that the system monitors the response times of each class over a period P and recomputes the allocations after each such period. We choose $P = 5s$ in our experiments. Unless specified otherwise, we choose a target response time of $d_i = 100$ms for each class and the threshold τ_i for discretizing the class states into the lo^-, lo, hi and hi^+ categories is set to 20ms.

We use a two types of workloads in our simulations: trace-driven and synthetic. We use NFS traces to determine the effectiveness of our methods for real-world scenarios. However, since a trace workload only represents a small subset of the operating region, we use a synthetic workload to systematic explore the state space.

We use portions of an NFS trace gathered from the Auspex file server at Berkeley [12] to generate the trace-driven workload. To account for caching effects, we assume a large LRU buffer cache at the server and filter out requests resulting in cache hits from the original trace; the remaining requests are assumed to result in disk accesses. The resulting NFS trace is very bursty and has a peak to average bit rate of 12.5.



(a) Average Response Time: Class 1 (b) Average Response Time: Class 2

Fig. 4. Behavior of the learning-based dynamic bandwidth allocation technique.

Our synthetic workload consist of Poisson arriving clients that read a randomly selected file. File sizes are assumed to be heavy-tailed; we assume fixed-size requests that sequentially read the selected file. By carefully controlling the arrival rates of such clients, we can construct transient overload scenarios (where a burst of clients arrive in quick succession).

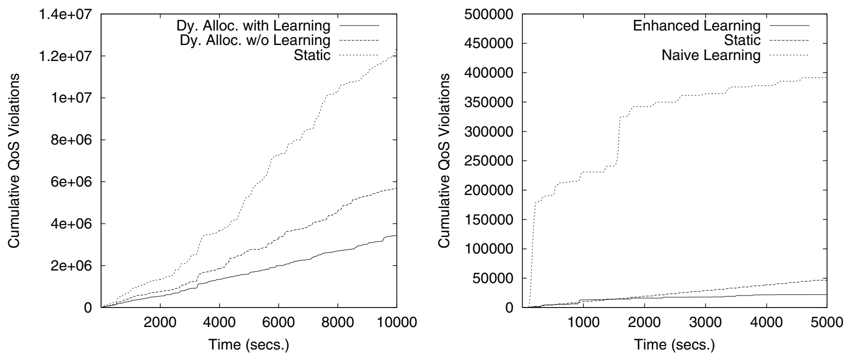
Next, we present our experimental results.

5.2 Effectiveness of Dynamic Bandwidth Allocation

We begin with a simple simulation experiment to demonstrate the behavior of our dynamic bandwidth allocation approach in the presence of varying workloads. We configure the system with two application classes. We choose an exponential smoothing parameter $\gamma = 0.5$, the learning step size $T = 5\%$ and the number of stored values per state $k = 5$. The target response time is set to 75ms for each class and the re-computation period was 5s. Each class is initially assigned 50% of the disk bandwidth.

We use a synthetic workload for this experiment. Initially both classes are assumed to have 5 concurrent clients each; each client reads a randomly selected file by issuing 4 KB requests. At time $t = 100s$, the workload in class 1 is gradually increased to 8 concurrent clients. At $t = 600s$, the workload in class 2 is gradually increased to 8 clients. The system experiences a heavy overload from $t = 700$ to $t = 900$s. At $t = 900s$, several clients depart and the load reverts to the initial load. We measure the response times of the two classes and then repeat the experiment with a static allocation of $(50\%, 50\%)$ for each class.

Figures 4 depicts the class response times. As shown the dynamic allocation technique adapts to the changing workload and yields response times that are close to the target. Further, due to the adaptive nature of the technique, the observed response times are, for the most part, better than that in the static allocation. Observe that, immediately after a workload change, the learning technique requires a short period of time to learn and adjust the allocations, and this temporarily yields a response time that is higher than that in the static case (e.g., at $t = 600s$ in Fig 4(b)). Also, observe that between $t = 700$ and $t = 900$ the system experiences a heavy overload and, as discussed in Case II of



(a) Trace Workload (b) Comparison with Simple Learning

Fig. 5. Comparison with Alternative Approaches

our approach, the dynamic technique resets the allocation of both hi^+ classes to their default values, yielding a performance that is identical to the static case.

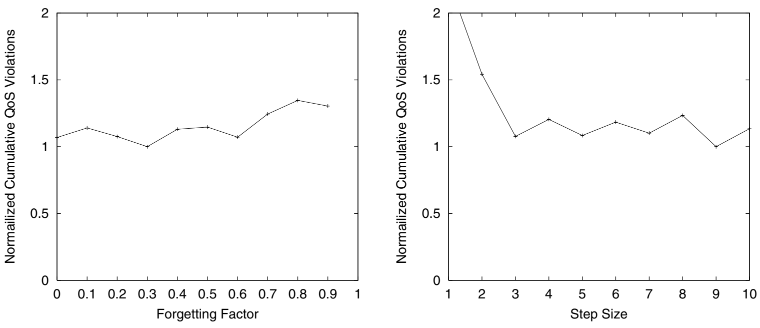
5.3 Comparison with Alternative Approaches

In this section, we compare our learning-based approach with three alternate approaches: (i) *static*, where the allocation of classes is chosen statically, (ii) *dynamic allocation with no learning*, where the allocation technique is identical to our technique but no learning is employed (i.e., allocations are left unchanged when all classes are underloaded or overloaded as in Cases I and II in Section 3.6, and in Case III bandwidth is reassigned from the least underloaded class to the most overloaded class in steps of T, but no learning is employed), and (iii) the *simple learning* approach outlined in Section 3.5.

We use the NFS traces to compare our enhanced learning approach with the static and the dynamic allocation techniques with no learning. We configure the system with three classes with different scale factors[3] and set the target responses time of each class to 100ms. The re-computation period is chosen to be 5s. We use different portions of our NFS trace to generate the workload for the three classes. The stripe unit size for the RAID-0 array is chosen to be 8 KB. We use about 2.8 hours of the trace for this experiment.

We run the experiment for our learning-based allocation technique and repeat it for static allocation and dynamic allocation without learning. In figure 5(a) we plot the cumulative $\sum sigma_{rt}^{+}$ (i.e., the cumulative QoS violations observed over the duration of the experiment) for the three approaches; this metric helps us quantify the performance of an approach in the long run. Not surprisingly, the static allocation techniques yields the worst performance and incurs the largest number of QoS violations. The dynamic allocation technique without learning yields a substantial improvement over the static approach, while dynamic allocation with learning yields a further improvement. Observe that the gap between static and dynamic allocation without learning *depicts the benefits of dynamic allocation over static*, while the gap between the technique without learning

[3] The scale factor scales the inter-arrival times of requests and allows control over the burstiness of the workload.



(a) Effect of Smoothing Parameter γ (b) Effect of Step Size t

Fig. 6. Impact of Tunable Parameters

and our technique *depicts the additional benefits of employing learning*. Overall, we see a factor of 3.8 reduction in QoS violations when compared to a pure static scheme and a factor of 2.1 when compared to a dynamic technique with no learning.

Our second experiment compares our enhanced learning approach with the simple learning approach described in Section 3.5. Most parameters are identical to the previous scenario, except that we only assume two application classes instead of three for this experiment. Figure 5(b) plots the cumulative QoS violations observed for the two approaches (we also plot the performance of static allocation for comparison). As can be seen, the naive learning approach incurs a larger search/learning overhead since it systematically searches through all possible actions. In doing so, incorrect actions that exacerbate the system performance are explored and actually worsen performance. Consequently, we see a substantially larger number of QoS violations in the initial period; the slope of the violation curve reduces sharply once some history is available to make more informed decisions. Consequently, during this initial learning process, a naive learning process under-performs even the static scheme; the enhanced learning technique does not suffer from these drawbacks, and like before, yields the best performance.

5.4 Effect of Tunable Parameters

We conduct several experiments to study how the choice of three tunable parameters affects the system behavior: the exponential smoothing parameter γ, the step size T and the history size k that defines the number of high reward actions stored by the system.

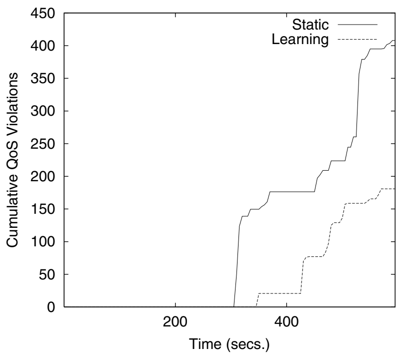
First, we study the impact of the smoothing parameter γ. Recall from Equation 1 that $\gamma = 0$ implies that only the most recent reward value is considered, while $\gamma = 1$ completely ignores reward values. We choose $T = 5\%$ and $k = 5$. We vary γ systematically from 0.0 to 0.9, in steps of 0.1 and study its impact on the observed QoS violations. We normalize the cumulative QoS violations observed for each value of γ with the minimum number of violations observed for the experiment. Figure 6(a) plots our results. As shown in the figure, the observed QoS violations are comparable for γ values in the range (0,0.6). The number of QoS violations increases for larger values of gamma—larger values of γ provide less importance to more recent reward values and consequently, result in larger QoS violations. This demonstrates that, in the presence of dynamically varying workloads, recent reward values should be given sufficient importance. We suggest choosing a γ between 0.3 and 0.6 to strike a balance between the recent reward values and those learned from past history.

Next, we study the impact of the step size T. We choose $\gamma = 0.5$, $k = 4$ and vary T from 1% to 10% and observe its impact on system performance. Note that a small value of T allows fine-grain reassignment of bandwidth but can increase the time to search for the correct allocation (since the allocation is varied only in steps of T). In contrast, a larger value of T permits a faster search but only permits coarse-grain reallocation. Figure 6(b) plots the normalized QoS violations for different values of T. As shown, very small values of T result in a substantially higher search overhead and increase the time to converge to the correct allocation, resulting in higher QoS violations. Moderate step sizes ranging from 3% to as large as 10% seem to provide comparable performance. To strike a balance between fine-grain allocation and low learning (search) overheads, we suggest step sizes ranging from 3-7%. Essentially, the step size should be sufficiently large to result in a noticeable improvement in the response times of borrowers but not large enough to adversely affect a lender class (by reclaiming too much bandwidth).

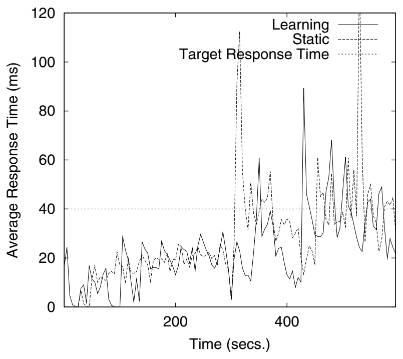
Finally, we study the impact of varying the history size k on the performance. We choose $\gamma = 0.5$, $T = 5\%$ and vary k from 1 to 10 (we omit the graph due to space constraints). Initially, increasing the history size results in a small decrease in the number of QoS violations, indicating that additional history allows the system to make better decisions. However, increasing the history size beyond 5 does not yield any additional improvement. This indicates that storing a small number of high reward actions is sufficient, and that it is not necessary to store the reward for every possible action, as in the naive learning technique, to make informed decisions. Using a small value of k also yields a substantial reduction in the memory requirements of the learning approach.

5.5 Implementation Experiments

We now demonstrate the effectiveness of our approach by conducting experiments on our Linux prototype. As discussed in Section 4, our prototype consists of a 8 disk system, configured as RAID-0 using the software RAID driver in Linux. We construct three volumes on this array, each corresponding to an application class. We use a a mix of three different applications in our study, each of which belongs to a different class: (1) *PostgreSQL database server:* We use the publicly available PostgreSQL database server version 7.2.3 and the *pgbench 1.2* benchmark. This benchmark emulates the TPC-B transactional benchmark and provides control over the number of concurrent clients as well as the number of transactions performed by each client. The benchmark generates a write-intensive workload with small writes. (2) *MPEG Streaming Media Server:* We use a home-grown MPEG-1 streaming media server to stream a 90 minute videos to multiple clients over UDP. Each video has a constant bit rate of 2.34 Mb/s and represent a sequential workload with large reads. (3) *Apache Web Server:* We use the Apache web server and the publicly available SURGE web workload generator to generate web workloads. We configure SURGE to generate a workload that emulates 300 time-sharing users accessing a 2.3 GB data-set with 100,000 files. We use the default settings in SURGE for the file size distribution, request size distributions, file popularity, temporal locality and idle periods of users. The resulting workload is largely read-only



(a) Cumulative QoS violations



(b) Database Server

Fig. 7. Results from our prototype implementation.

and consists of small to medium size reads. Each of the above application is assumed to belong to separate application class. To ensure that our results are not skewed by a largely empty disk array, we populated the array with a variety of other large and small files so that 50% of the 144GB storage space was utilized. We choose $\gamma = 0.5, T = 5\%$, $k = 5$ and a recomputation period $P = 5s$. The target response times of the three classes are set to 40ms, 50ms and 30ms, respectively.

We conduct a 10 minute experiment where the workload in the streaming server is fixed to 2 concurrent clients (total I/O rate of 4.6 Mb/s). The database server is lightly loaded in the first half of the experiment and we gradually increase the load on the Apache web server (by starting a new instance of the SURGE client every minute; each new client represents 300 additional concurrent users). At $t = 5$ minutes, the load on the web server reverts to the initial load (a single SURGE client). For the second half of the experiment, we introduce a heavy database workload by configuring pgbench to emulate 20 concurrent users each performing 500 transactions (thereby introducing a write-intensive workload).

Figure 7(a) plots the cumulative QoS violations observed over the duration of the experiment for our learning technique and the static allocation technique. As shown, for the first half of the experiments, there are no QoS violations, since there is sufficient bandwidth capacity to meet the needs of all classes. The arrival of a heavy database workload triggers a reallocation in the learning approach and allows the system to adapt to this change. The static scheme is unable to adapt and incurs a significantly larger number of violations. Figure 7(b) plots the time-series of the response times for the database server. As shown, the adaptive nature of the learning approach enables it to provide better response times to the database server. While the learning technique provides comparable or better response time than static allocation for the web server, we see that both approaches are able to meet the target response time requirements (due to the light web workload in the second half, the observed response times are also very small). We observe a similar behavior for the web server and the streaming server. As mentioned before, learning could perform worse at some instants, either if it is exploring the allocation space or due to a sudden workload change, and it requires a short period to readjust the allocations. In figure 7(b) this happens around $t = 400$ s when learning performs worse than static, but the approach quickly takes corrective action and gives better performance.

Overall, the behavior of our prototype implementation is consistent with our simulation results.

5.6 Implementation Overheads

Our final experiment measures the implementation overheads of our learning-based bandwidth allocator. To do so, we vary the number of disks in the system from 50 to 500, in steps of 50, and measure the memory and CPU requirements of our bandwidth allocator. Observe that since we are constrained by a 8 disk system, we emulate a large storage system by simply replicating the response times observed at a single disk and reporting these values for all emulated disks. From the perspective of the bandwidth allocator, the setup is no different from one where these disks actually exist in the system. Further, since the allocations on each disk is computed independently, such a strategy accurately measures the memory and CPU overheads of our technique. We assume that new allocations are computed once every 5s.

We find that the CPU requirement for our approach to be less than 0.1% even for systems with 500 disks, indicating that the CPU overheads of the learning approach is negligible. The memory overheads of the allocation daemon are also small, with the percentage of memory used on a server with 1 GB RAM varies (almost linearly) from 1 MB (0.1 %) for a 50 disk system to 7 MB (0.7 %) for a 500 disk system.

Finally, note that the system call overheads of querying response times and conveying the new allocations to the disk scheduler can be substantial in a 500 disk system (this involves 1000 system calls every 5 seconds, two for each disk). However, observe that, the bandwidth allocator was implemented in user-space for ease of debugging; the functionality can be easily migrated into kernel-space, thereby eliminating this system call overhead. Overall, our results demonstrate the feasibility of using a reinforcement learning approach for dynamic storage bandwidth allocation in large storage systems.

6 Related Work

Recently, the design of self-managing systems has received significant research attention. For instance, the design of workload monitoring and adaptive resource management for data-intensive network services has been studied in [9]. The design of highly-dependable ("self-healing") Internet services has been studied [15].

From the perspective of storage systems, techniques for designing self-managing storage have been studied in [2,4]. The design of such systems involves several sub-tasks and issues such self-configuration [2,4] , capacity planning [8], automatic RAID-level selection [5], initial storage system configuration [3] , SAN fabric design [22] and online data migration [13]. These efforts are complementary to our work which focuses on automatic storage bandwidth allocation to applications with varying workloads.

Dynamic bandwidth allocation for multimedia servers has been studied in [20]. Whereas the approach relies on a heuristic, we employ a technique based on reinforcement learning. Several other approaches ranging from control theory to online measurements and optimizations can also be employed to address this problem. While no such study exists for storage systems, both control theory [1] and online measurements and optimizations [6,16] have been employed for dynamically allocating resources in web servers. Utility-based optimization models for dynamic resource allocation in server clusters have been employed in [11]. Feedback-based dynamic proportional share allocation to meet real-rate disk I/O requirements have been studied in [17]. While many feedback-based methods involve approximations such as the assumption of a linear relationship between resource share and response time, no such limitation exists for reinforcement learning—due to their search-based approach, such techniques can easily handle non-linearity in system behavior. Alternative techniques based on linear programming also make the linearity assumption, and need a linear objective function which is minimized; such a linear formulation may not be possible or might turn out to be inaccurate in practice. On the other hand, a hill-climbing based approach can handle non-linearity, but can get stuck in local maxima.

Finally, reinforcement learning has also been used to address other systems issues such as dynamic channel allocation in cellular telephone systems [19] and adaptive link allocation in ATM networks [14].

7 Concluding Remarks and Future Work

In this paper, we addressed the problem of dynamic allocation of storage bandwidth to application classes so as to meet their response time requirements. We presented an approach based on reinforcement learning to address this problem. We argued that a simple learning-based approach is not practical since it incurs significant memory and search space overheads. To address this issue, we used application-specific knowledge to design an efficient, practical learning-based technique for dynamic storage bandwidth allocation. Our approach can react to dynamically changing workloads, provide isolation to application classes and is stable under overload. Further, our technique learns online and does not require any *a priori* training. Unlike other feedback-based models, an additional advantage of our technique is that it can easily handle complex non-linearity in the system behavior. We implemented our techniques into the Linux kernel and evaluated it using prototype experimentation and trace-driven simulations. Our results showed that (i) the use of learning enables the storage system to reduce the number of QoS violations by a factor of 2.1 and (ii) the implementation overheads of employing such techniques in operating system kernels is small. Overall, our work demonstrated the feasibility of using reinforcement learning techniques for dynamic resource allocation in storage systems. As part of future work, we plan to explore the use of such techniques for other storage management tasks such as configuration, data placement, and load balancing.

References

1. T. Abdelzaher, K.G Shin and N. Bhatti. Performance Guarantees for Web server End-Systems: A Control Theoretic Approach. *IEEE Transactions on Parallel and Distributed Systems.* 13(1), January 2002.
2. G. A. Alvarez et al. Minerva: An Automated Resource Provisioning Tool for Large-scale Storage Systems. *ACM Transactions on Computer Systems*(to appear). *Technical report HPL-2001-139, Hewlett-Packard Labs*, June 2001.

3. E. Anderson et al. Hippodrome: Running Circles Around Storage Administration. *In FAST'02, Monterey, CA*, pp. 175–188, Jan. 2002.
4. E. Anderson et al. Ergastulum: An Approach to Solving the Workload and Device Configuration Problem. *HP Laboratories SSP technical memo HPL-SSP-2001-05*, May 2001.
5. E. Anderson, R. Swaminathan, A. Veitch, G. A. Alvarez and J. Wilkes. Selecting RAID levels for Disk Arrays. *In FAST'02, Monterey, CA*, pp. 189–201, January 2002.
6. M. Aron et al. Scalable Content-aware Request Distribution in Cluster-based Network Servers. *Proceedings of the USENIX 2000 Annual Technical Conference, San Diego, CA*, June 2000.
7. P. Barham. A Fresh Approach to File System Quality of Service. *In Proceedings of NOSSDAV' 97, St. Louis, Missouri*, pages 119–128, May 1997.
8. E. Borowsky et al. Capacity planning with phased workloads. *In Proceedings of the Workshop on Software and Performance (WOSP'98), Santa Fe, NM*, October 1998.
9. 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, Arizona*, March 1999.
10. J. Carlström and E. Nordström. Reinforcement learning for Control of Self-Similar Call Traffic in Broadband Networks. *Proceedings of the 16th International Teletraffic Congress, ITC'16*, P. Key., D. Smith (eds.), Elsevier Science, Edinburgh, Scotland, 1999.
11. J. Chase et al. Managing Energy and Server Resources in Hosting Centers. *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.
12. M. Dahlin et al. A Qualitative Analysis of Cache Policies for Scalable Network File Systems. *In Proceedings of the ACM SIGMETRICS '94*, May 1994.
13. C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: Online Data Migration with Performance Guarantees. *In FAST'02, Monterey, CA*, pp. 219–230, January 2002.
14. E. Nordström and J. Carlström. A Reinforcement Learning Scheme for Adaptive Link Allocation in ATM Networks. *IWANNT '95*, J. Alspector, T.X. Brown, pp. 88–95, Lawrence Erlbaum, Stockholm, Sweden, 1995.
15. D.A. Patterson et al. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. *UC Berkeley Computer Science Technical Report UCB//CSD-02-1175*, March 15, 2002.
16. P. Pradhan, R. Tewari, S. Sahu, A. Chandra and P. Shenoy. An Observation-based Approach Towards Self-managing Web Servers. *In Proceedings of ACM/IEEE Intl Workshop on Quality of Service (IWQoS), Miami Beach, FL*, May 2002.
17. 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.
18. P. Shenoy and H. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. *In Proceedings of ACM SIGMETRICS '98, Madison, WI*, pp. 44–55, June, 1998.
19. S. Singh and D. Bertsekas. Reinforcement Learning for Dynamic Channel Allocation in Cellular Telephone Systems. With D. Bertsekas. *In NIPS 10*, 1997.
20. V. Sundaram and P. Shenoy. Bandwidth Allocation in a Self-Managing Multimedia File Server. *Proceedings of the Ninth ACM Conference on Multimedia, Ottawa, Canada*, Oct. 2001.
21. R. S. Sutton and A G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.
22. J. Ward, M. O'Sullivan, T. Shahoumian, and J. Wilkes. Appia: Automatic Storage Area Network Design. *In FAST'02, Monterey, CA*, pp. 203–217, January 2002.
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.