Towards End-to-End Quality of Service: Controlling I/O Interference in Shared Storage Servers

Gokul Soundararajan and Cristiana Amza

Department of Electrical and Computer Engineering
University of Toronto

Abstract. Due to the imperative need to reduce the costs of management, power and cooling in large data centers, operators multiplex several concurrent applications on each physical server of a server farm connected to a shared network attached storage. Determining and enforcing per-application resource quotas on the fly in this context poses a complex resource allocation and control problem spanning many levels including the CPU, memory and storage resources within each physical server and/or across the server farm. This problem is further complicated by the need to provide end-to-end Quality of Service (QoS) guarantees to hosted applications.
In this paper, we introduce a novel approach towards controlling application interference for resources in shared server farms. Specifically, we design and implement a minimally intrusive method for passing application-level QoS requirements through the software stack. We leverage high-level per-application requirements for controlling I/O interference between multiple database applications, by QoS-aware dynamic resource partitioning at the storage server. Our experimental evaluation, using the MySQL database engine and OLTP benchmarks, shows the effectiveness of our technique in enforcing high-level application Service Level Objectives (SLOs) in shared server farms.

1 Introduction

As the costs of management, power and cooling in large data centers become prohibitive, automated *server consolidation* techniques for better resource usage while providing differentiated Quality of Service (QoS) to applications become increasingly important. With *server consolidation*, several concurrent applications are multiplexed on each physical server of a server farm connected to consolidated network attached storage (see Figure 1). Such architectures are common in large data centers and consist of multiple levels of software, including web and application servers, database servers, operating systems and the storage server at the lowest level. The challenge for providing QoS to applications in these environments lies in the complexity of the dynamic resource partitioning problem for avoiding application interference at multiple levels, i.e., for CPU, memory and storage.

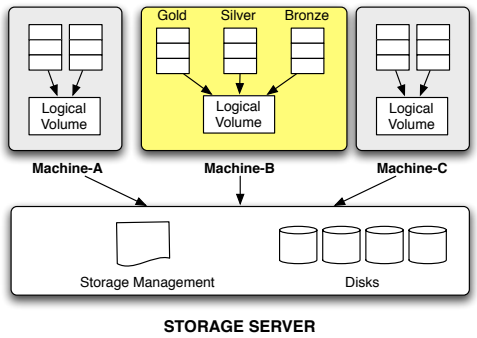


Fig. 1. Modern enterprise architecture: Server farm with resource consolidation.

Previous work on dynamic resource partitioning in shared server environments focuses on partitioning a single resource within a single software tier at a time. Specifically, resource virtualization through virtual machine monitors (VMMs) has been used in both generic server systems [3] and database systems [14, 15] to enforce per-application CPU quotas. Similarly, memory quota enforcement has been studied within the buffer pool of a database system running several applications [4, 5]. Finally, several techniques have been studied for partitioning the I/O bandwidth between applications within the storage server [11, 12, 19]. However, the above approaches fall short of providing effective resource partitioning due to the following two reasons.

The first reason is that application QoS is usually expressed as a high-level Service Level Objective (SLO), e.g., desired latency or throughput, not as per-resource priorities or quotas. There is currently no automatic mechanism to assign the relative priority levels or resource quotas for applications corresponding to a high-level application metric. A dynamic approach to resource allocation is clearly more desirable than extensive off-line profiling in modern data center environments, where the set of co-scheduled applications, and/or the type and availability of hardware resources may change frequently and unpredictably.

The second reason that prevents current approaches from providing effective resource partitioning is the absence of coordination between different resource controllers. This absence of coordination might lead to situations where local goals may conflict with each other, or with the high-level per-application goals. For instance, the operating system may optimize fairness in thread scheduling across applications, while the storage server may optimize I/O latency. Each resource controller optimizes local goals, oblivious to the goals of other resource controllers, and to the per-application SLO's. There is little or no previous work on correlating priority or quota enforcement across several resources or software components.

To address the dynamic resource allocation problem in consolidated server environments, we introduce a novel technique for controlling application interfer-

ence. Our technique determines per-application resource quotas on the fly, with minimal application instrumentation. To achieve this, we monitor application-level metrics relative to SLOs periodically and pass these as application utility values down through all levels of the software stack i.e., from the DBMS to the OS running on each physical server in a server farm, and then to the shared storage server.

The monitored application-level metrics are utilized by a coordinated distributed learning technique, with one adaptive controller per software component. Each resource controller uses a reinforcement learning algorithm, called *learning automata* (LA) [13], for resource allocation. Specifically, each LA controller employs a feedback loop to dynamically converge towards a resource partitioning setting that minimizes the perceived penalties for all applications.

Though our technique is general enough to be applied for partitioning all shared resources, at all tiers, in this paper, we focus on dynamically partitioning the storage bandwidth. Towards this goal, we implement our technique in a prototype that enforces coordinated resource quotas per application at two levels: i) at the operating system I/O scheduler within each physical server of the server farm and ii) at the shared storage level. Our prototype implementation shows that our approach can be integrated in existing environments and applications with minimal changes to interfaces between components.

Specifically, we modify the Linux kernel and the Network Block Device (NBD) protocol, a network block protocol that is bundled with the Linux kernel, to allow passing the application-level utility on I/O calls, and to implement our learning and I/O scheduling algorithms. Our technique is sufficiently flexible to enforce resource quotas and to change them dynamically, for different applications, but also per application thread within the same application e.g., to enforce differentiated QoS for performance-critical transactions or queries.

We perform experiments on a cluster of dual processor servers connected to a storage server with external direct attached storage. We use the MySQL database engine and two applications: DBT-2 and the ORION (Oracle IO Numbers) storage utility. DBT-2 is a classic OLTP workload similar to TPC-C. Orion emulates part of the common I/O workload of the Oracle database server. We run experiments in several configurations where instances of the two applications share physical servers as well as the storage server. We show convergence to the per-application quotas that meet the high-level application SLO's for each application when using our coordinated dynamic learning technique.

The remainder of this paper is structured as follows. Section 2 describes the role of each software component in servicing I/O requests and the motivation to use end-to-end resource partitioning. Section 3 describes the architecture of our system and introduces our coordinated learning and our I/O bandwidth partitioning technique. Section 4 describes our prototype implementation. Section 5 presents our benchmarks and experimental platform, while Section 6 presents the results of our experiments on this platform. Section 7 discusses related work and Section 8 concludes the paper.

2 Background and Motivation

Enterprise storage servers (Figure 1) provide an abstraction of a single large logical storage device carved into several logical volumes. An application, like a database system or file system, mounts the logical volumes and uses the underlying storage. Within this storage hierarchy, we focus on the following two levels of control: (1) the OS I/O scheduler, which schedules I/O requests from a storage client to the underlying storage device, and (2) the storage server I/O scheduler, which manages bandwidth allocations to different logical volumes.

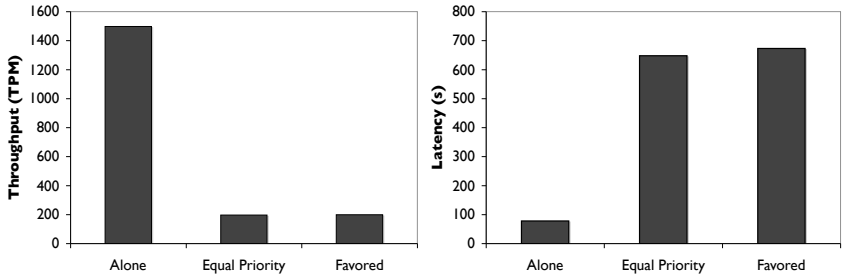
The interactions between the storage client and server travel through the operating system's *block layer*. The *block layer* maps logical to physical accesses on block devices e.g., in a RAID. It provides a wide range of functionality from request sorting and merging, prefetching, to I/O scheduling i.e., reordering requests to optimize the disk seek time. Due to this commonly used optimization objective, physically sequential I/O will be preferentially scheduled, typically regardless of its high-level application SLO. To counter starvation, implementations of I/O scheduling either attach a deadline for every request or provide fairness among several streams. However, these approaches are typically unaware of application SLOs. Similar to the operating system I/O scheduler, the storage server schedules I/O requests from different logical volumes. While the operating system I/O scheduler attempts to minimize seek times, the storage server I/O scheduler controls when each workload's request is sent to the disk firmware e.g., in order to meet a pre-specified I/O *latency* [12], but oblivious to the high-level application SLO.

As we can see from our description of the storage hierarchy described above, storage bandwidth allocations are influenced by both the operating system and the storage server, in an uncoordinated, SLO unaware, and possibly conflicting manner. In the following, we show through a motivating experiment that unpredictably large performance degradation can occur for co-scheduled applications due to I/O interference, whether or not CPU priorities are enforced at the OS level.

2.1 Motivating Example

Using current I/O schedulers in existing operating systems, we show that the performance of an application can be severely affected when paired with another I/O intensive process, whether or not we enforce per-application CPU priorities at the operating system level.

We run DBT-2, a TPC-C like workload, on MySQL, concurrently with OLTP-A, an online transaction processing (OLTP) workload, generated using the ORION (Oracle IO Numbers) tool. We configure DBT-2 to use 200 warehouses, resulting in a database size of 64GB. We provide additional details of our workloads in Section 6. In our experiments, we run MySQL/InnoDB on direct-attached storage and show the effects of I/O interference between applications. In our experimental setup, we use either the `cfq` scheduler, recently added to the Linux kernel, which attempts to provide fair queuing among several processes, or the



(a) Throughput (b) Latency

Fig. 2. Co-scheduling DBT-2/OLTP-A on Direct-Attached Storage with CFQ scheduling

more traditional `deadline` scheduler in Linux, which primarily targets minimizing I/O seek time. Neither the `cfq` scheduler, nor the `deadline` scheduler support enforcement of application SLOs.

Figure 2 shows the I/O interference between DBT-2 and OLTP-A when using the `cfq` scheduler at the operating system. We see that, when DBT-2 runs alone, it achieves 1498 TPM (transactions per minute) and its 90^{th} percentile latency is 78 seconds. However, when DBT-2 is co-scheduled with OLTP-A, there is a significant slowdown. DBT-2's throughput is only 13% of its throughput running in isolation and the latency is 8.3x the original latency. In an attempt to achieve better performance for DBT-2, we set the CPU nice levels for DBT-2 to -10 (high priority) and OLTP-A to +10 (low priority) and re-run the experiment. We see a very small gain in DBT-2's performance. The throughput increases slightly from 196 TPM to 198 TPM.

The interference effect is even more pronounced when using the traditional `deadline` scheduler in Linux. In this case, DBT-2's throughput when co-scheduled with OLTP-A is only 2.3% of its throughput when running in isolation i.e., 80 TPM compared to 3391 TPM. As before, setting the DBT-2 process to a higher priority, by using the UNIX nice utility, does not significantly alleviate the problem.

These results show that there is currently no method of enforcing I/O requirements of applications at the operating system level. Furthermore, there is no method of communicating application SLO requirements and enforcing them at the storage server. Since both the OS and the storage server perform I/O scheduling in a per-application QoS oblivious manner, current architectures are unable to enforce end-to-end quality of service. As we have shown, this results in potentially high performance degradation for the high priority application.

In this paper, we address these issues by providing a method of transmitting application SLO requirements throughout the storage hierarchy. This allows the individual I/O controllers at each level to determine the bandwidth allocations dynamically.

3 Providing End-to-End QoS via Coordinated Learning

In this section, we describe our approach to dynamic resource allocation in a server farm with network attached storage. Our objective is to allocate each application enough resources (i.e., bandwidth) to meet its SLO. Towards this, we use coordinated learning to determine resource quotas dynamically, at two levels in the system: the OS and the storage server. In the following, we first introduce the overall architecture of our system and an overview of our approach. Then, we describe our coordinated learning and dynamic quota enforcement algorithm in detail. Finally, we discuss the trade-offs made in our design.

3.1 Architecture and Problem Statement

The architecture of our system is presented in Figure 1. We show the storage server hosting a number of virtual devices connected to several physical servers i.e., machines A, B, and C. Each physical server hosts a number of application classes, e.g., *gold*, *silver*, and *bronze* hosted on Machine-B in the Figure.

In this environment, the problem of resource allocation can be described as follows. For k servers hosting n application classes connected to s virtual volumes hosted on the storage server, we need to find the following proportions in order to meet the specified SLOs: i) We need to find proportions $P_{S_1}, P_{S_2}, \ldots, P_{S_s}$ for enforcing disk bandwidth partitioning among the workload for the virtual volumes at the storage server. and ii) At each machine $m \in \{1, 2, \ldots, k\}$, we need to determine the proportions $P_{m_1}, P_{m_2}, \ldots, P_{m_n}$ for scheduling the respective requests to the virtual device at the level of the OS I/O scheduler (e.g., $P_{B_{gold}}$, $P_{B_{silver}}$, and $P_{B_{bronze}}$ for the OS on Machine-B in Figure 1.

Finding an optimal solution to this problem is challenging since there is no clear mapping from the specified SLOs to disk bandwidth. As such, we use adaptive machine learning techniques as described next.

3.2 Overview of Approach

Towards achieving the specified SLOs, we embed resource controllers at the OS and at the storage server. All resource controllers use a learning algorithm for dynamic resource partitioning at its level. Specifically, each resource controller changes its own per-workload proportions dynamically, converging to a local solution based on application-level feedback values.

We coordinate learning between the OS and the storage server through a token-passing scheme. The learners at the two levels take turns in making actions and observing the application feedback. In this way, each level can observe the application feedback based only on its actions, thus converging to a solution. We ensure convergence to a stable global resource partitioning solution for the different learners by using the same feedback metric for learners at both the OS and the storage server levels. This application-level feedback metric, called *Deviance from Target* (dft), is periodically monitored for each application. The most recent dft is then passed from the application level through all levels of the

software stack, including the OS, to the storage server on each I/O call of the corresponding application. Finally, each resource controller enforces the learned resource partitioning through quanta based scheduling for its workloads. Modifications to existing interfaces between components are minimal; all information exchanged between the two levels is piggybacked on regular communication.

In the following, we introduce the high-level application metric we use for coordinated learning (the dft metric). Next, we introduce the learning algorithm employed at each resource controller and the coordination between resource controllers. Finally, we explain the quanta-based scheduling algorithm used by each resource controller for enforcing the resource partitioning.

3.3 Deviance From Target (DFT) Metric

We use a single high-level application-level metric, called *Deviance from Target* (dft), for guided learning at all resource controllers. The dft represents the utility to the service provider from meeting the service level objective (SLO) of the corresponding application. This utility is typically mapped directly to an expected monetary reward (or penalty) for hosting a particular application and it may combine two factors: i) a performance indicator i.e., the relative distance of a pre-specified application metric, such as transaction throughput, or latency from a contracted SLO value over time and ii) the contracted client priority or class for the corresponding application e.g., gold/silver/bronze or best-effort.

Without loss of generality, for the purposes of this paper, we use as performance indicator a number that indicates the deviation from expected application performance, where a 0.0 value corresponds to *target achieved*, a positive value means we have exceeded the objective and a negative value means a violation of the contracted performance, hence a penalty for the service provider. For example, in order to compute the dft for a particular high priority OLTP application, we periodically sample the transactions completed. Then we compute the normalized distance between the average transaction throughput value over the last sampling interval and the contracted/expected throughput value (the SLO). To produce the dft, this value would be typically weighted to include the priority class. For simplicity, for the purposes of this paper, we use only two classes: priority and best effort. For a best effort application we always provide a dft feedback value of 0.0 regardless of the performance indicator. For a priority application, we provide its performance indicator as the dft feedback.

Finally, we also support assigning different dft values for different threads, transactions or queries inside an application. Specifically, we support selectively tagging fine-grained application contexts with the overall dft value of an application, while all other I/O from that application should be classified as best effort. For example, a database application may signal that a DBMS application thread carries its overall utility rather than a DBMS statistics logger thread; alternatively, the application may assign all its utility to a key transaction type e.g., a payment transaction.

In the following, we describe our learning algorithm at each resource controller. We then introduce a lightweight and minimally intrusive technique to

coordinate the multiple controllers implemented at different levels in order to provide end-to-end QoS.

3.4 Learning at Each Resource Controller

We determine the workload proportions dynamically using a reinforcement learning algorithm [18]. In reinforcement learning, the learning agent learns how to use various *actions* to maximize a numerical *reward*. We use a simple reinforcement learning algorithm named *learning automata* (LA) [13].

Learning automata are adaptive decision-making devices that operate in unknown environments. A learning automaton has a finite set of actions and each action has a certain probability (unknown to the automaton) of getting rewarded by the environment of the automaton. The aim is to learn to choose the optimal action (i.e. the action with the highest probability of being rewarded) through repeated interactions with the system. If the environment is sufficiently stationary during the learning period, the iterative process of interacting with the environment in the LA algorithm is guaranteed to converge to the optimal solution [13]. We use a linear reward-penalty learning automata, where an automaton can probabilistically choose one of r actions $\{a_1, a_2, \ldots, a_r\}$ with associated probabilities $\{p_1, p_2, \ldots, p_r\}$ respectively. Let $p(k)$ denote the probability of an action to be taken at iteration k and suppose action a_i is taken at iteration k.

The result of an action a_i is mapped to a range between 0.0 and 1.0, where 0.0 represents the maximum positive feedback and 1.0 represents the maximum negative feedback. The feedback for the k^{th} action is represented using the variable $f(k)$. The probabilities for taking each action are updated as follows. The probability p_i corresponding to action a_i is updated to:

$$p_i(k+1) = p_i(k) - \beta f(k) p_i(k) + \alpha(1 - f(k))(1 - p_i(k)) \tag{1}$$

All other actions, a_j where $i \neq j$ are updated to:

$$p_j(k+1) = p_j(k) + f(k)(\frac{\beta}{r-1} - \beta p_j(k)) - \alpha(1 - f(k)) p_j(k) \tag{2}$$

The parameters α and β scale the reward and penalty. Typically, $\alpha > \beta$ for faster convergence.

We describe how we adapt the LA learning algorithm to enable dynamic allocations in our controller. The goal at each controller is to minimize the sum of the squared deviations from 0.0 (*error*) for the dft of all applications. For example, if the storage server was hosting s virtual volumes with each virtual volume hosting n applications, then the *error* (e) would be computed as

$$e = \sum_{i=1}^{s} \sum_{j=1}^{n} [dft_{i,j}]^2 \tag{3}$$

Each controller dynamically determines proportions between its workloads, i.e., between applications in the operating system and between virtual devices

in the storage server scheduler, with the objective of minimizing the error (e). For instance, consider a controller which schedules two workloads at the storage server. Such a controller will simply have to determine the proportion $0 \leq P \leq 1.0$, such that one workload receives a fraction P and the other workload receives $1 - P$ of the resource. For that particular controller, in order to determine P, we define a number of actions for the LA learning algorithm representing bandwidth allocations. The controller's action sets the proportions by picking from a collection of discrete choices. In our example here, a possible collection of choices might be {10/90, 30/70, 50/50, 70/30, 90/10}.

At each learning iteration, the controller first measures the current error, e_{cur}, in the system. Then, it probabilistically selects an action to take. For example, the controller may select the action corresponding to enforcing a proportion of 50/50. After selecting an action, the controller waits until the effects of its action are visible, for either a fixed time interval or a fixed number of requests. It then evaluates the application-level feedback, computes the new *error* value, e_{new}, and updates the variable $f(k)$ with a new value between 0.0 and 1.0, depending on the perceived benefit of its action. Finally, the controller updates the probabilities corresponding to taking each action using the new value of $f(k)$ in the formulas above.

3.5 Coordinated Learning

While all controllers in our system have the same goal, i.e., to optimize the dft error for all applications, each learns iteratively through trial and error. Thus, if all learners actuate their proportions in parallel, the feedback received by each learner is the result of actions taken by all controllers, not just by itself. To enable accurate feedback, hence convergence towards an end-to-end solution, we coordinate the multiple learners in the hierarchy using a simple token passing scheme. We thus let either the OS-level controllers or the storage server controller learn at a given time, while keeping the proportions fixed at the other level. Token requests and replies are passed on regular requests and replies between the two levels. Whenever holding the token, a learner takes a number of actions actuating its per-workload proportions and observes the application feedback on incoming requests.

3.6 Enforcing Proportions through Quanta-based Scheduling

We enforce proportions by using quanta-based scheduling [19] at both the OS and storage resource controllers. Specifically, we partition a scheduling period into time intervals and assign intervals to workloads to meet their respective proportions. For example, let the scheduling period be 100 milliseconds with 100 slices. If two applications, A and B require equal proportions, then, each would be given exclusive access to storage for 50 milliseconds in every 100 milliseconds scheduling period. Scheduling based on time quanta allows for a good combination of enforcing proportions between workloads as well as taking advantage of the usual storage optimizations for per-workload locality. This is because when

only one workload is allowed to run during a time interval, during that time, both the OS/storage I/O schedulers can optimize disk seeks with the usual techniques e.g., using elevator scheduling and also exploit the disk cache for that workload.

3.7 Discussion

In this section we discuss the trade-offs in our scheduling technique. We then present a theoretical argument for convergence to a global optimal solution for our end-to-end approach.

Trade-offs in Scheduling Technique. While quanta-based scheduling ensures that each workload receives a share of the disk bandwidth, there is an inherent tradeoff between using coarse-grained versus fine-grained scheduling intervals, hence quanta. At the limit, the scheduler can simply not use time quanta at all, and issue requests proportionally from each workload. Using large quanta may waste disk bandwidth if insufficient requests from the respective workload are available to the scheduler during a particular quantum. On the other hand, as mentioned before, using coarse-grained quanta has the advantage of reducing the potential disk seeks and cache conflicts caused by switching between multiple workloads.

We note that in many practical cases, the adaptivity inherent in our approach will naturally alleviate penalties, by self-regulating the quanta granularity. For example, assume that a sequential workload suffers due to increased disk seeks when interleaved with a random-access workload at the storage. If these penalties are significant, they will be reflected in the application's high-level metrics. Hence, the sequential workload will automatically receive a larger proportion of I/O bandwidth. The larger bandwidth allocation will implicitly translate into a *larger quanta*.

Global Convergence to an Optimal Solution. Our coordinated learning technique will converge towards a state with the minimum penalties achievable for the applications, hence for the service provider, if the application behavior and environment does not substantially change *during learning*.

When using multiple learners with a common feedback signal, as in our case, each environment state is determined by a combination of actions from all learners. In this case, the environment states form a composite environment which is referred to as Markovian Switching Environment. In such an environment, it can be theoretically shown [13] that a variable-structure automata with ergodic techniques, such as the linear reward-penalty learning automata we use, converges to the optimal set of actions by the multiple learners within a margin of error due to the continuous learning and exploratory nature of LA controllers.

The ideal solution, where all application `dft`'s are 0.0 may be, however, unattainable, e.g., because of insufficient overall I/O bandwidth, and dynamic provisioning of additional resources may become necessary.

4 Prototype Implementation

In this section, we describe our prototype implementation for passing high-level application metrics through the software stack to the storage controller, and our virtual storage controller implementation.

4.1 Overview of Prototype Implementation

We embed our LA controller into Linux and our virtual storage prototype. We leverage the Network Block Device (NBD) code available with Linux for this purpose. NBD is a standard storage access protocol, similar to iSCSI, supported by Linux. NBD provides a method for a storage client (in our case MySQL) to communicate with a storage server over the network; specifically, NBD provides a pair of client/server modules, which run on the same physical machines as the storage client/server, respectively.

We implement a Linux-based virtual storage prototype, which we deploy on top of our commodity storage (RAID) firmware. We modify the existing *client* and *server* NBD protocol processing modules in order to pass high-level application metrics to our LA storage bandwidth controller. Specifically, we piggyback the application's performance (`dft`), the application identifier, and a learning token on the I/O call path. Our storage controller enforces bandwidth quota allocations, maps virtual to physical block accesses and issues the appropriate I/O requests to disk.

4.2 Code Changes

We instrument MySQL to capture the application-level metrics of interest, periodically, and to compute the *dft* metric relative to a predefined SLO for each application context. For example, for DBT-2, we monitor transaction throughput as application level metric and for Orion we use latency, which are the standard QoS metrics for these applications.

For every I/O call made from MySQL on behalf of the application, we add arguments to the corresponding system call and pass the application context identifier and the periodically updated *dft* metric for that application context. Context identifiers are assigned in such a way to be unique cluster-wide. In order to support differentiated QoS for fine-grained and/or dynamic application-level contexts, e.g., per application thread, or per-transaction, we also add a new system call, `ioprio_context()`, to the Linux kernel. `ioprio_context()` signals the beginning and end of an application context. We add corresponding system calls in MySQL, reusing pre-existing begin and end markers in the application structure e.g., for transaction begin and commit or thread creation and destruction. We modify the Linux kernel and the NBD packet format to tag each I/O call with the application-level information and pass this information through the respective software layers. In addition, for the coordinated learning algorithm, we piggyback the learning token on request and reply NBD packets.

4.3 I/O Scheduling Implementation

When a workload is given a quantum, we first determine the number of requests we can issue to disk such that they complete within the workload's quantum. To compute this value, we maintain an exponentially weighted average of the disk service time and the application's concurrency level. Using these two values, we compute the number of requests that can be issued per workload such that all requests finish within the quantum. First, we issue requests that were enqueued while waiting for the quantum to begin. Then, we issue requests that arrive during the scheduling quantum. We stop issuing requests if we determine that by issuing a request, we will exceed the workload's quantum. In this case, new requests will be enqueued as we wait for the requests to return from disk before the next quantum begins.

5 Experimental Methodology

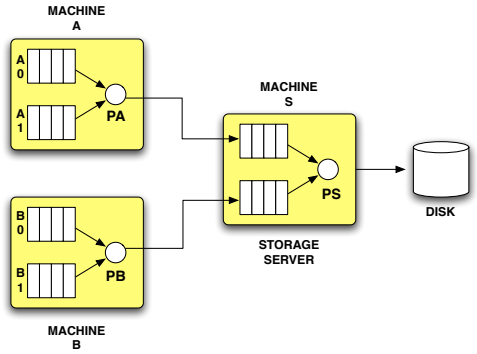


Fig. 3. Experimental Multi-tier System with Shared Storage

We create a multi-tier system with shared storage using NBD, as shown in Figure 3. We use three machines: a storage server (S) and two application servers (A and B). In this system, we can run 4 applications, A_0, A_1, B_0, and B_1. The storage server (S) serves two virtual block devices which are mounted by machines A and B, respectively. The applications A_0 and A_1 share one virtual block device and B_0 and B_1 share the other. In addition, each machine runs a LA based controller that determines the bandwidth allocation for the two incoming streams. Machine A determines P_A, the fraction of the bandwidth allocated to A_0. Conversely, A_1 receives $(1-P_A)$ bandwidth. Similarly, Machine B determines P_B and the storage server S determines P_S.

The application servers are Dell PowerEdge SC1425 with dual Intel Xeon processors running Ubuntu Linux 6.06 with our modifications, and connected by

Gigabit Ethernet. The storage server is a Dell PowerEdge PE1950 with 4 Intel Xeon processors running at 3Ghz and 3GB of memory. The storage server is connected to an external direct attached storage with 15 10K RPM SAS hard drives. The attached storage is configured using RAID-0. We benchmark the direct attached storage using ORION and found it provides 800 IOPS for our microbenchmark OLTP-A. Our NBD based storage server increases the latency by at most 10%.

We use MySQL/InnoDB and configure it to use a raw device and a buffer pool of 512 MB. We use ORION (Oracle IO Numbers) as a I/O load generator. ORION is a calibration tool released by Oracle to benchmark different storage architectures for database workloads. It allows the user to set different parameters like block size, read/write ratio, and number of outstanding I/Os. By changing the parameters, one can generate different types of database workloads. We set these parameters to generate an OLTP-like workload classified with equal amount of reads and writes, many random I/O accesses (16KB block size) and some large I/O (1MB blocks).

5.1 Benchmarks

OLTP-A: OLTP-A is an OLTP-*like* workloads we generate using the ORION tool. It is characterized by many random I/O accesses of 16KB and some large I/O of 1MB. The read/write ratio is 50%. We configure ORION to have 100 outstanding small I/O and 10 outstanding large I/O. OLTP-A issues I/O to a 64GB raw partition.

DBT-2: DBT-2 is an OLTP workload derived from TPC-C benchmark [16]. It simulates a wholesale parts supplier that operates using a number of warehouse and sales districts. Each warehouse has 10 sales districts and each district serves 3000 customers. The workload involves transactions from a number of terminal operators centered around an order entry environment. There are 5 main transactions for: (1) entering orders, (2) delivering orders, (3) recording payments, (4) checking the status of the orders, and (5) monitoring the level of stock at the warehouses. We scale DBT-2 by using 256 warehouses and the footprint of the database is 60GB. In our experiments, we simulate 1000 users connected to the system.

6 Experimental Results

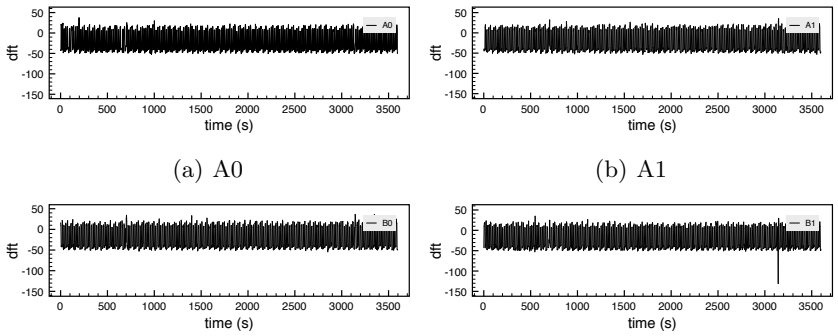
We present an experimental evaluation of our end-to-end I/O bandwidth allocation technique. All results are obtained on our experimental configuration described in the previous section. We first evaluate our learning technique for enforcing end-to-end resource allocations. We then show the benefits of coordinated versus uncoordinated learning in two sharing scenarios, using the ORION and DBT-2 benchmarks.

6.1 Benefits of Coordinated Learning

We show the benefits of coordinated versus uncoordinated learning with two sharing scenarios. First, we run four instances of OLTP-A. Next, we co-schedule DBT-2 with three instances of OLTP-A. For both scenarios, we compare both coordinated and uncoordinated learning with two ideal scenarios, where proportions are set manually for either i) one resource controller or ii) both resource controllers. In more detail, we evaluate four schemes:

1. **Optimal Settings:** We set all proportions manually to the optimal configuration.
2. **Single Storage Learner:** We set the proportions manually in the OS schedulers (P_A and P_B) but we determine P_S at the storage through learning.
3. **Uncoordinated Learning:** We let all controllers find the optimal values in parallel.
4. **Coordinated Learning:** We enable our token passing algorithm to coordinate the controllers. For uncoordinated and coordinated learning, we initialize the probabilities in each controller to $\frac{1}{5} = 0.2$ such that each action is equally likely.

In all experiments, we plot the *dft* (deviation from target) versus time. If $dft < 0$ this signifies that the application did not meet its SLO and a $dft > 0$ indicates that the application performed better than its SLO. Ideally, the $dft = 0$ throughout the duration of the experiment.

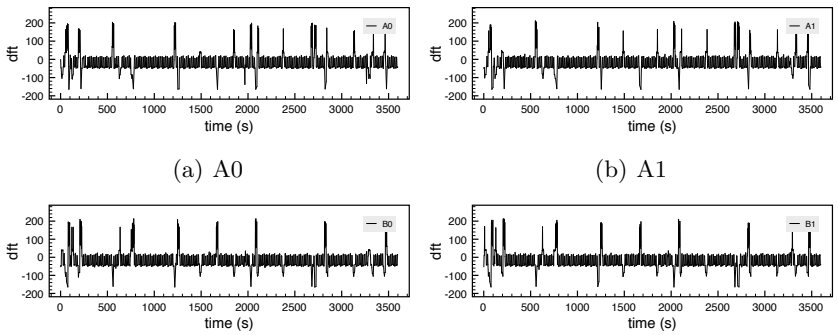


(a) A0 (b) A1

(c) B0 (d) B1

Fig. 4. OLTP-A performance using Optimal Settings

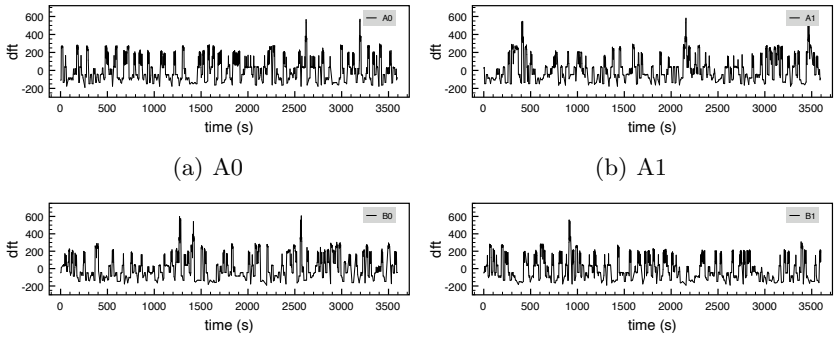
OLTP-A: In the first experiment, we run 4 instances of OLTP-A, two on each of our physical servers in our experimental setup. Since all workloads are identical, the optimal configuration is 0.5 at P_A, 0.5 at P_B and 0.5 at P_S. Figure 4 shows the performance of OLTP-A when allocations were optimally chosen. We



(a) A0 (b) A1

(c) B0 (d) B1

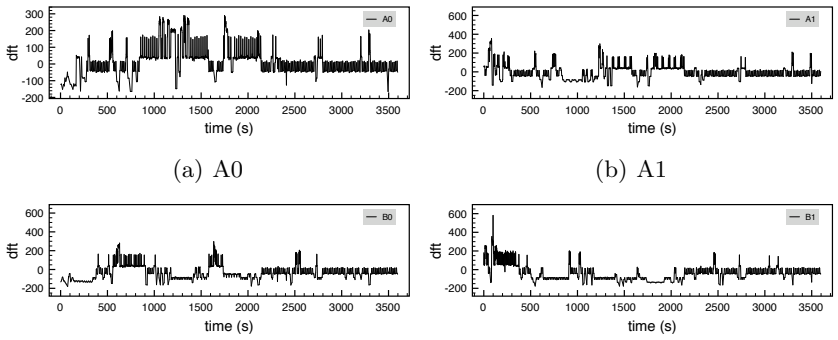
Fig. 5. OLTP-A performance using Single Learner



(a) A0 (b) A1

(c) B0 (d) B1

Fig. 6. OLTP-A performance using Uncoordinated Learning



(a) A0 (b) A1

(c) B0 (d) B1

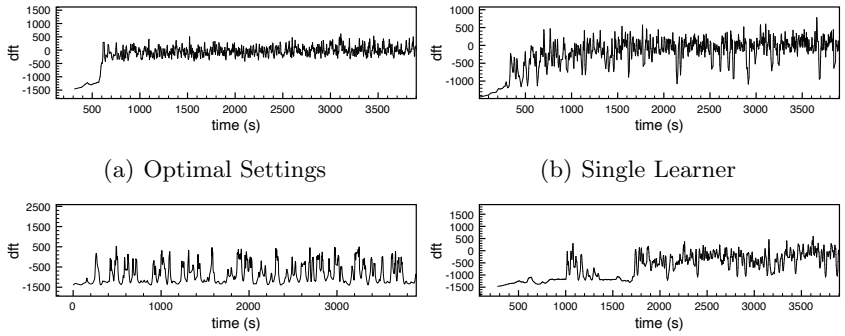
Fig. 7. OLTP-A performance, using Coordinated Learning

show the results of all four OLTP-A instances. Since there is no learning stage, the allocations are met from the beginning of the experiment. In the second experiment, we fixed $P_A = 0.5$ and $P_B = 0.5$ and we used our controller to determine P_S. Figure 5 shows that the storage controller initially explores the solution space. At about the 300 second mark, the controller converges to the optimal action. After convergence, each application is on target with slight variations due to the adaptive and exploratory nature of the controller.

As shown in Figure 6, the uncoordinated controllers are not able to converge within the duration of the experiment. Given the duration of the experiment of almost one hour, each SLO violation shown in the figure is of substantial amplitude, on the order of minutes in duration, and occurs roughly every 5-10 minutes. Hence, the QoS provided is unacceptable, and performance of all applications is poor. In contrast, as Figure 7 shows, the token passing algorithm allows the controllers to converge to an optimal allocation. At about the 2000 second mark, the three controllers arrive at the optimal solution and the performance of each application reaches its target. We observe that the token passing algorithm slows the learning process since each controller can run only while holding the token.

DBT-2/OLTP-A: We run DBT-2 with one OLTP-A workload on one physical server and 2 OLTP-A workloads on the second server. For DBT-2, we set the SLO at 80% percent its throughput running alone in the system (as measured in transactions/minute, TPM) and we classify it as a high priority application. We classify all OLTP-A workloads as best effort. With these requirements, our goal is to satisfy the performance demands of DBT-2 and divide the remaining bandwidth to the OLTP-A workloads.

We run the same four experiments as before. In the first experiment, we set the values of $P_A = 0.9$, $P_B = 0.5$ and $P_S = 0.9$ such that DBT-2 receives $P_A * P_S = 0.9 * 0.9 = 0.81 = 81\%$ of the available storage bandwidth. Figure 8(a) shows that, after an initial warmup stage, DBT-2 quickly reaches the target performance and stays on target for the duration of the experiment.



(a) Optimal Settings (b) Single Learner

(c) Uncoordinated Learning (d) Coordinated Learning

Fig. 8. DBT-2 performance

In the second experiment, we fixed $P_A = 0.9$ and $P_B = 0.5$ but we allow the controller to determine the optimal value of P_S. As shown in Figure 8(b), the storage controller arrives at the optimal configuration after the initial learning stage. This experiment also highlights the resilience of our controller. DBT-2 has an initial warmup stage before it begins to run the measurement stage. In the warmup stage, the workload uses fewer clients thus placing a smaller demand on the system. Therefore, the controller chooses a proportion that is optimal for the warmup stage. When DBT-2 begins the measurement stage, the controller adapts by selecting a different proportion that is optimal for the measurement stage of DBT-2. The results show that, even with the dynamic nature of DBT-2, the controller is able to adapt and arrive at the optimal configuration by the 1000 second mark of the experiment.

As before, the uncoordinated learners are not able to converge to the optimal configuration during our experiment (as shown in Figure 8(c). This results in poor performance for DBT-2, which does not converge to its target performance. In contrast, Figure 8(d) shows that with coordinated learning, the controllers are able to converge to the ideal solution at about the 2000 second mark of the experiment and are able to meet the DBT-2 performance target. The highest probability actions at each level of control after convergence, are close to the ideal proportion settings: 90/10 for DBT-2/OLTP-A, and 50/50 for OLTP-A/OLTP-A for the proportions at the two OS controllers, respectively and 90/10 for the proportions at the storage controller. Thus, while the DBT-2 performance target is achieved, requests from the best effort OLTP-A applications are also serviced.

7 Related Work

Resource allocation is a well known technique for improving system performance. Traditionally, resource scheduling has been achieved using either a *priority-based*

mechanism or a *quanta-based* mechanism. Under priority-based mechanisms, applications with low priority are prone to starvation. This makes such mechanisms inappropriate when the objective is to provide per application QoS guarantees. In contrast to priority-based mechanisms, quanta-based scheduling mechanisms guarantee that each transaction acquires a fair portion of the shared resource e.g., as in *lottery scheduling* where processes are assigned *tickets* proportional to their share [21]. However, in this work, administrators need to manually specify the proportions for each application. *Real-rate scheduling*, is another policy with similarities to our own, in which the applications provide the OS scheduler a notion of *progress* through timestamps [9]. Using this information, the real-rate scheduler employs a feedback loop to determine resource requirements and specifies them to a proportion-period scheduler.

Dynamic allocation of the disk bandwidth has been studied to provide QoS at the storage server. Just like in our prototype, SLEDS [8], Façade [12], SFQ [10], and Argon [19] place a scheduling tier above the existing disk scheduler which controls the I/Os issued to the underlying disk. Argon [19] uses a quanta-based scheduler, while SLEDS [8] uses a *leaky-bucket* filter to throttle I/Os from clients exceeding their given fraction. Similarly, SFQ dynamically adjusts the deadline of I/Os to provide fair sharing of bandwidth. Furthermore, Cello [17] and YFQ [6] build QoS-aware disk schedulers, which make low-level scheduling decisions that strive to minimize seek times, as well maintain quality of service. All previous work in this area has studied methods on disk bandwidth allocation at a single level, either at the operating system level or at the storage level. We have shown that layering of several controllers leads to oscillation, hence suboptimal behavior. Through our context aware approach, we coordinate the controllers at both the operating system and at the storage server to provide QoS guarantees. Moreover, our technique is general and can easily be extended to coordinated resource partitioning of different resources e.g., CPU and disk, and/or resource controllers for the same resource located within different tiers.

Resource allocation has also been studied in database systems. Current implementations of DBMS rely on simple policies like Round-Robin for scheduling transaction access to CPU [7, 1]. More sophisticated adaptive algorithms providing per-class response time goals for queries of multiple classes have been studied for dynamic buffer pool partitioning [4, 5]. On the other hand, I/O scheduling as well as resource allocation to improve application defined metrics have not been studied in detail in database systems.

Finally, *resource containers* and *Virtual Machine Monitors (VMM)* provide mechanisms to enforce resource allocation [2]. For example, the VMWare ESX server employs memory allocation algorithms to facilitate the execution of multiple virtual machines on a system and offers a performance guarantee to each [20]. However, I/O performance isolation at the storage level, which is the main bottleneck in modern enterprise environments, is currently not guaranteed with these mechanisms.

8 Conclusion

We study techniques for enforcing end-to-end Quality of Service for applications in shared server farms. We introduce a unifying approach for controlling application interference for resources at all levels of the storage stack. Our approach uses coordinated learning based on the degree of achievement of high-level per-application service level objectives.

We implement our approach with minimal changes to existing interfaces in a state-of-the-art shared infrastructure using commodity software and hardware components. We focus on dynamically partitioning I/O bandwidth at two levels: the operating system I/O scheduler and the shared storage scheduler.

We evaluate coordinated versus uncoordinated learning as well as coordinated learning versus the optimal manually set configuration for enforcing I/O bandwidth allocations. We show experimentally, using industry standard benchmarks, that our technique converges towards the optimal configuration and is effective in enforcing high-level application SLOs at the storage server.

References

1. R. K. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *VLDB*, pages 385–396, 1989.
2. G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, pages 45–58, 1999.
3. P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.
4. K. P. Brown, M. J. Carey, and M. Livny. Managing memory to meet multiclass workload response time goals. In *VLDB*, pages 328–341, 1993.
5. K. P. Brown, M. J. Carey, and M. Livny. Goal-oriented buffer management revisited. In H. V. Jagadish and I. S. Mumick, editors, *SIGMOD Conference*, pages 353–364. ACM Press, 1996.
6. J. L. Bruno, J. C. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *ICMCS, Vol. 2*, pages 400–405, 1999.
7. M. J. Carey, R. Jauhari, and M. Livny. Priority in DBMS Resource Scheduling. In *VLDB*, pages 397–410, 1989.
8. D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. In *SRDS*, pages 109–118. IEEE Computer Society, 2003.
9. A. Goel, J. Walpole, and M. Shor. Real-rate scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 434–441. IEEE Computer Society, 2004.
10. P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. Netw.*, 5(5):690–704, 1997.
11. A. Gulati, A. Merchant, and P. J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. In L. Golubchik, M. H. Ammar, and M. Harchol-Balter, editors, *SIGMETRICS*, pages 13–24. ACM, 2007.

12. C. R. Lumb, A. Merchant, and G. A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *FAST*, 2003.
13. K. S. Narendra and M. A. L. Thathachar. *Learning Automata: An Introduction*. Prentice Hall, Englewood Cliffs, NJ, 1989.
14. O. Ozmen, K. Salem, M. Uysal, and M. H. S. Attar. Storage workload estimation for database management systems. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *SIGMOD Conference*, pages 377–388. ACM, 2007.
15. P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *EuroSys*, pages 289–302. ACM, 2007.
16. F. Raab. TPC-C - The Standard Benchmark for Online transaction Processing (OLTP). In J. Gray, editor, *The Benchmark Handbook*. Morgan Kaufmann, 1993.
17. P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. *SIGMETRICS Perform. Eval. Rev.*, 26(1):44–55, 1998.
18. R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. ISBN 0-262-19398-1, auch siehe `http://www.cs.ualberta.ca/~sutton/book/ebook/the-book.html`.
19. M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *FAST*, pages 61–76, Berkeley, CA, USA, 2007. USENIX Association.
20. C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI*, 2002.
21. C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *OSDI*, pages 1–11, 1994.