LA-UR- 11-00409

*Approved for public release; distribution is unlimited.*

| | |
|---|---|
| *Title:* | QoS Support for Users of I/O-intensive Applications using Shared Storage Systems |
| *Author(s):* | Xuechen Zhang, Marion Kei Davis, Song Jiang |
| *Intended for:* | ICS Conference 2011 |

Los Alamos
NATIONAL LABORATORY
EST. 1943



Form 836 (7/06)

# QoS Support for End Users of I/O-intensive Applications using Shared Storage Systems

Xuechen Zhang
ECE Department
Wayne State University
5050 Anthony Wayne Drive
Detroit, MI, 48202, USA
xczhang@wayne.edu

Kei Davis
CCS Division
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
kei.davis@lanl.gov

Song Jiang
ECE Department
Wayne State University
5050 Anthony Wayne Drive
Detroit, MI, 48202, USA
sjiang@eng.wayne.edu

## ABSTRACT

I/O-intensive applications are becoming increasingly common on today's high-performance computing systems. While performance of compute-bound applications can be effectively guaranteed with techniques such as space sharing or QoS-aware process scheduling, it remains a challenge to meet QoS requirements for end users of I/O-intensive applications using shared storage systems because it is difficult to differentiate I/O services for different applications with individual quality requirements. Furthermore, it is difficult for end users to accurately specify performance goals to the storage system using I/O-related metrics such as request latency or throughput. As access patterns, request rates, and the system workload change in time, a fixed I/O performance goal, such as bounds on throughput or latency, can be expensive to achieve and may not lead to a meaningful performance guarantees such as bounded program execution time.

We propose a scheme supporting end-users' QoS goals, specified in terms of program execution time, in shared storage environments. We automatically translate the users' performance goals into instantaneous I/O throughput bounds using a machine learning technique, and use dynamically determined service time windows to efficiently meet the throughput bounds. We have implemented this scheme in the PVFS2 parallel file system and have conducted an extensive evaluation. Our results show that this scheme can satisfy realistic end-user QoS requirements by making highly efficient use of the I/O resources. The scheme seeks to balance programs' attainment of QoS requirements, and saves as much of the remaining I/O capacity as possible for best-effort programs.

## Keywords

Shared Storage System, QoS, PVFS2

## 1. INTRODUCTION

Provision of QoS guarantees to high-performance applications, such as climate and weather forecasting [12], modeling financial data [4], combustion simulation [36], human genome analysis [28], and even chess-playing [14], can be critical to the success of the services provided to their users. For example, for an application predicting the timing for a hurricane reaching land, users usually have firm deadlines for receiving the prediction results because of the potential for life and property losses if the deadline is passed. When it is not feasible to dedicate a powerful parallel computing system and I/O subsystem to a single application, a mechanism for ensuring service quality in terms of execution deadlines in a shared execution environment should be provided by a production parallel computing installation.

It is relatively easy to satisfy the QoS requirement for a compute-bound application by allocating a dedicated partition of compute nodes (space sharing), or by ensuring a sufficient fraction of CPU time allocation (time sharing). However, if the application is I/O-intensive, or I/O services constitute a substantial portion of its total execution time, it is much more difficult to satisfy the application's QoS requirements on a parallel computing system using a shared storage system. There are several challenges.

First, for end users, execution time or response time of an application is usually the first choice for QoS specification. For compute-bound applications the time can generally be translated to a number of compute nodes or CPU time slices in a relatively straightforward manner. For I/O-intensive applications, users currently must determine the I/O service quality commensurate with the required execution time, assuming the computational resource is fully supplied, and then reserve resources in the storage system according to the derived I/O service requirement. However, users may not have the expertise or knowledge to derive the service requirements. For example, users may not know the actual amount of data requested or the service time spent on data access in a program's run.

Second, currently QoS assurance, or resource reservation, is provided by maintaining an upper bound on request latency, or a lower bound on throughput, specified for each application. Maintaining a single latency bound or a throughput bound cannot guarantee user-observed performance and may not allow the shared storage system to be used efficiently for the following reasons. (1) Request size can be highly variable. Requests of different sizes may need different latency bounds to reflect applications' true demands on storage resources imposed by their QoS requirements. (2) I/O requests can be very bursty. A bound on latency, or request response time, that includes request waiting time, may

have to vary accordingly. If the bound is specified in terms of throughput, a single bound would not serve the end-user's performance goal. If the average throughput is taken as the bound, the application's demand may not be well served when the request arrival rate is high because the bound is effectively under-estimated, and an application's reserved resource may be under-utilized when the arrival rate is low. Though taking the highest possible throughput as a bound can safely meet an application's QoS requirement, the storage resource could be excessively over-provisioned and the global efficiency of the storage system significantly compromised. (3) The spatial locality of requests can vary. Spatial locality describes the sequentiality of requested data on the storage device. Hard-disk-based storage is highly sensitive to this workload characteristic because significant seek time is usually required when accessing non-sequential data. Spatial locality is a highly elusive property that is difficult for users to characterize. If an average latency or throughput bound is used regardless of spatial locality, the application may be over-provisioned for its sequential accesses, and the I/Osystem could be overly stressed for its random accesses. All of these dynamics exhibited in a program's run show that using a single I/O performance bound for QoS guarantee is less than ideal. On the other hand, in general it is too difficult to manually derive bounds associated with the aforementioned factors.

Third, a shared storage system usually consists of multiple disks, or multiple data servers, where files are striped. While each data server or even each disk has its own I/O scheduling, the I/O performance experienced by an individual program is the aggregate effect of scheduling its requests at different data servers. To maintain a throughput bound for a program, common wisdom to improve disk efficiency is to dedicate a time window for exclusively serving requests from the same program to minimize the interference from requests issued by other concurrently running programs. However, implementing service windows for a program independently at each data server will usually lead to unacceptably low disk efficiency. When file data are striped over multiple data servers, contiguous requests from a program may be spread over multiple servers. Therefore, if the requests are synchronous, a data server may have a relatively long wait between completing the program's current request and receiving its next request, as intervening requests are serviced by other servers that may be busy serving other programs' requests. Because of the long wait the disk head must move to serve requests from other programs instead of idly waiting for the program's next request. Thus the disks cannot be dedicated to serving one program at a time when using non-work-conserving I/O schedulers such as *anticipatory* [15], leaving the disk heads to constantly seek between disk regions accessed by different programs. This makes resource allocation according to QoS requirements yet more difficult, and causes undue interference among programs.

We propose a scheme supporting end-user-specified QoS goals in terms of program execution time. To derive instantaneous I/O performance goals from the specified execution time for a program, we first capture the storage performance characteristics—the relationship between access patterns and instantaneous throughout—both by running the program of interest, and with a probing program, in a storage environment where either the entire storage system, or a fixed fraction of it, is dedicated for use in the profiling. The measured characteristics are then used to train machine-learning models so that, in an actual run of the program on a shared storage environment, the model can estimate the instantaneous I/O throughput required to meet the specified deadline according to the observed access pattern. The I/O resource is then dynamically allocated to the program according to estimated instantaneous throughputs. To effectively allocate the resource, we coordinate the I/O scheduling at different data servers for dedicated service to one or multiple programs to implement QoS goals presented to the storage system by the trained machine learning models.

In summary, we make the following contributions.

- We propose a QoS performance interface for end users to conveniently specify their QoS requirements in terms of program execution times. We use a machine learning technique to automatically characterize the storage system and convert the end-user QoS requirements into instantaneous I/O performance goals at run time.
- We coordinate the I/O scheduling at different data servers to facilitate effective QoS-aware resource allocation. We take both I/O QoS requirements and overall storage system efficiency into account by opportunistically dedicating disk services to requests from one or multiple programs. While the ultimate goal is to meet user-defined deadlines, we allow reduced resource commitment to a program when the storage system is overloaded and seek to make up corresponding performance loss when the system has unused capacity.
- We have implemented the design in the PVFS2 parallel system as a prototype, named *U-Shape*, that allows *U*sers to *Shape* the I/O scheduling implicitly to meet their QoS goals specified with execution times. Our extensive evaluation of U-Shape shows that it can efficiently implement end-user-specified performance goals and provide strong isolation among programs sharing a common storage system.

The rest of this paper is organized as follows. Section 2 uses a motivating example to demonstrate the efficacy of automatic derivation of I/O performance goals. Section 3 gives a detailed description of the design of U-Shape. Section 4 presents and analyzes experiment results. Section 5 describes related work and Section 6 concludes.

## 2. A MOTIVATING EXAMPLE

Here we demonstrate how conventional I/O performance requirements, such as average I/O throughput, can mislead the scheduling policy of a storage system and violate end-users' QoS goals, and give an initial indication of U-Shape's potential benefits.

Our experimental cluster consists of 13 nodes, four configured as compute nodes and the other nine as data servers, managed by a PVFS2 parallel file system, using PVFS2's default striping size of 64KB. One of data servers also serves as the metadata server for the file system. More details of the platform are given in Section 4.

Two MPI programs used as microbenchmarks in the experiments. The first, *full-seq*, sequentially reads a 10GB file using collective I/O, wherein each MPI process reads 64KB per request. The second, *seq-rand*, is designed to simulate I/O access patterns alternating between sequential access

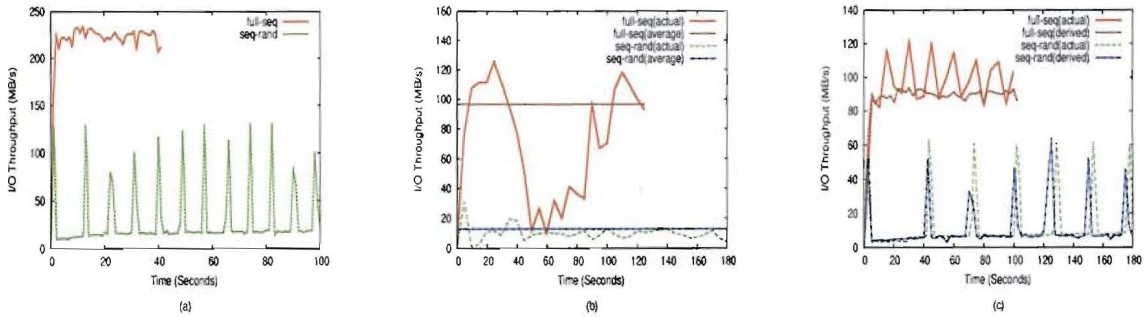



**Figure 1: Throughput of programs *full-seq* and *seq-rand* in different environments: (a) Each program runs exclusively on the storage system. (b) Both programs run together, both using average throughput as performance goals. The averages for each are marked "average". (c) Both programs run together using U-Shape dynamically-derived throughput as performance goals. The derived throughputs are marked "derived".**

(such as reading a large data file) and random access (such as searching index files). Specifically, the program uses collective I/O to sequentially read 500MB from the file, and then randomly reads another 500MB from the same file. The size of requests from each process is 64KB. This access pattern repeats until the entire 10GB file has been read. Each program runs as eight processes, two per compute node.

Figure 1(a) shows the instantaneous throughput within each one-second time window for each of the programs when it has dedicated use of the data servers. For *full-seq* the system delivers an average throughput of 242MB/s, close to the peak throughput of the system (270MB/s), and its execution time is 42.3s. The execution time of *seq-rand* is much greater (320s) and the average throughput is 32MB/s.

Assuming users of the programs expect reduced resource committed to each of their programs when both run concurrently on a shared storage system, such that they can tolerate up to 2.5-times slowdown, they set their performance goals as execution times of 106s and 800s for programs *full-seq* and *seq-rand*, respectively. If average throughput is used as the QoS requirement on the storage system, the system must consistently maintain 96.6MB/s and 12.8MB/s throughputs for *full-seq* and *seq-rand*, respectively. In the experiment we also run a best-effort program in the background, whose requests are served only when the system has excess capacity. Run concurrently, the execution times for *full-seq* and *seq-rand* are 126s and 1009s, exceeding their user-required times by 19% and 26%, respectively.

When we use the instantaneous throughputs derived by the machine learning model in the U-Shape scheme as the I/O performance goals, the execution times are 104s and 804s, which are very close to the QoS requirements of 106s and 800s, respectively.

To investigate the disparity between the two performance interfaces, we plot instantaneous throughputs of the two concurrently running programs using constant average throughput (Figure 1(b)), and using throughput derived by the trained machine learning models to prescribe QoS requirements (Figure 1(c)). As Figure 1(b) shows, the throughput of *full-seq* exhibits periodicity resonating with changes in *seq-rand*'s access pattern. When *seq-rand* issues sequential requests (from 0s to 38s and from 100s to 124s), *full-seq* mostly receives throughput near its required 96.6MB/s, and *seq-rand* can mostly meet its requirement. However, when *seq-rand* is issuing random requests (from 38s to 100s), *full-seq*'s throughput plummets and *seq-rand* also receives lower throughput. This is because maintaining the required 12.8MB/s throughput—the average of both sequential and random throughputs—when the access pattern is fully random, demands disproportionate resources and causes the system to be over-committed. Consequently, *full-seq*'s throughput is severely degraded. When *seq-rand* issues sequential requests the required throughput, which is in part determined by considering random access, appears to be too conservative in demanding system resources to meet its deadline. The result is that both programs miss their deadlines.

In the U-Shape scheme, the derived throughput of a program is always proportional to that for its dedicated run. Instead of maintaining a fixed throughput, U-Shape attempts to maintain a fixed resource allocation calculated from the end-users' QoS requirement. Accordingly, when access is sequential a higher throughput goal is used, and a lower throughput goal is used when access is random. This allows the system to be able to consistently meet varying throughput requirements as shown in Figure 1(c).

## 3. THE DESIGN OF U-SHAPE

Next we describe the design of the U-Shape scheme, whose objective is to meet QoS goals specified by end users in terms of program execution time through efficiently implementing automatically derived I/O performance requirements in a shared storage system. To this end, U-Shape first takes as inputs the end-users' QoS goals and I/O behaviors collected in profiling runs of their programs to train a machine learning model. It then uses the models on-line to derive I/O throughput bounds for each time interval (*epoch*) and seeks to achieve the bounds through effective resource allocation.

### 3.1 Derivation of Instantaneous Throughput Bounds

In this work we assume that a user has reserved a certain dedicated computing resource for running his/her program in the hope of achieving bounded execution time. For

a program with substantial I/O activity the objective can be achieved only when the program's I/O service time is also bounded. From the perspective of the storage system, the share of resource committed to the program should be bounded as there may be other programs concurrently running with their own demands on I/O service quality. As shown in the motivating example, the resource demand can differ by an order of magnitude between sequential and random access to deliver the same throughput. Therefore, we need to determine throughput bounds at different times during a program's execution such that (1) the resource demand for achieving the throughput is bounded; and, (2) the program's total I/O time is bounded. For this purpose we take a profiling run of the program with a fixed share (which may be 100%) of the storage system service time dedicated to serving its requests. The results of the profiling run give the fraction of execution time that is I/O time for a given share of storage service.

We also collect spatial locality information in each epoch and its corresponding throughput during a profiling run. Specifically, we collect on-disk data locations, for every request $R_i$ from the program, in terms of logical block number $LBN_i$ and size $S_i$, as well as its service time $ServTime_i$ at each data server. Because we evenly partition the program execution period into a series of time epochs, its requests are grouped by epoch. We quantify spatial locality of the requests in each epoch with two metrics: average distance $AvgDist$ between two consecutive requests and average request size $AvgSize$, assuming there are $n$ requests in the epoch $[R_1, R_2, \ldots, R_n]$, as

$$AvgDist = \frac{\sum_{i=2}^{n}[LBN_i - (LBN_{i-1} + S_{i-1})]}{n-1},$$

$$AvgSize = \frac{\sum_{i=1}^{n} S_i}{n}.$$

In addition, the throughput $B_{profile}$ in an epoch in the profiling run is calculated as

$$B_{profile} = \frac{\sum_{i=1}^{n} S_i}{\sum_{i=1}^{n} ServTime_i}.$$

Note that in the calculation of the throughput we do not count compute times between consecutive requests, so the metric is not affected by request arrival rate. This is the throughput that contributes to the program's execution time $T_{profile}$ in the profiling run, which is the sum of compute time $T_{comp}$ and I/O time $T_{profile\_io}$. Assume that the end-user-required program execution time for a real run is $T_{real}$, which is the sum of compute time $T_{comp}$ and I/O time $T_{real\_io}$ in the run. We assume that the program has its dedicated compute resource, so the compute time does not change across runs. We also assume that requests are synchronous. Therefore, the ratio $\alpha$ between the two I/O times is

$$\alpha = \frac{T_{real\_io}}{T_{profile\_io}} = \frac{T_{real} - T_{comp}}{T_{profile} - T_{comp}}.$$

To ensure that the storage resource requested by the program is bounded, we consistently apply the ratio to the throughput of each epoch to obtain the required throughput bound $B_{real}$ for a real run, i.e.,

$$B_{real} = \frac{B_{profile}}{\alpha}.$$

We use statistics on the locality and the derived throughput bound for real runs to train the widely used machine learning model CART (classification and regression tree) [31]. In the training session we treat $(AvgDist, AvgSize)$ of an epoch as the epoch's feature vector. We feed the feature vector and corresponding throughput bound $B_{real}$ of each epoch to the model so that it can 'learn' the association between spatial locality and throughput. The training algorithm recursively constructs a decision tree, starting with the root node, and grows the tree by adding child nodes to a leaf node at each step until its mis-prediction error rate measured in the cross-validation step is smaller than a predefined error threshold. For a given feature vector, a well-trained model is expected to provide a relatively accurate estimate of throughput that is suitable as a bound in real runs. A trained model is deployed in the storage system to predict instantaneous throughput. Figure 2 illustrates the major steps in training the model and using it to derive throughput bounds.

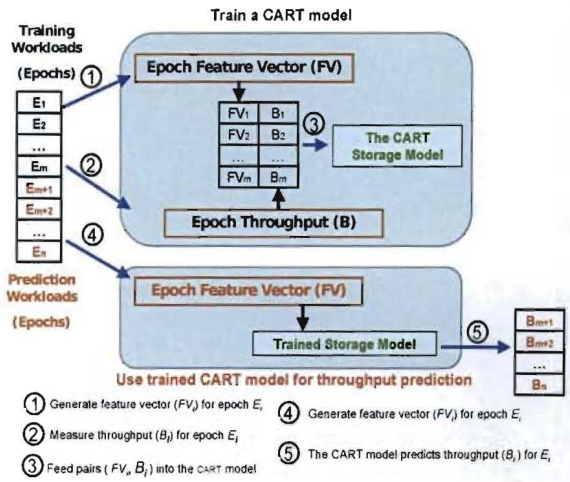



**Figure 2: The major steps involved in training a CART model with a feature vector for requests in each time epoch, and the steps for deriving throughput bounds for epochs of known request locality.**

With the use of profiling runs we make the assumption that I/O intensity and access locality are usually consistent across different runs of a program. The assumption is often valid, especially when the bulk of the input remains constant. This is common, for example, in the usage of many scientific applications wherein ensembles are run and only initial conditions are varied.

For the trained model to make accurate predictions the training data must provide sufficient coverage of potential workload characteristics. To this end we use a synthetic I/O workload in addition to real workloads in the profiling runs to generate the training data. We developed a synthetic I/O request generator to create a workload covering a spectrum of request sizes, spatial localities, and read/write ratios. In the workload, request size is uniformly distributed between one block (4KB) and a threshold size (128KB). Requests exceeding the threshold are split into a series of se-

quential requests of 128KB or smaller before feeding them into the model. Spatial locality is considered to be the probability that two successively served requests are contiguous on disk. In the generation of synthetic I/O workloads, we use a predefined probability to dictate whether two successive requests are sequential. The ratio of read and write requests is uniformly distributed between 100% reads and 100% writes. In a real run, we run a daemon at each data server to track the spatial locality of requests to the server and regenerate the feature vector periodically. The daemon then runs the model with an epoch's locality to derive the required throughput bound $B_{real}$. On the assumption that locality does not frequently make abrupt changes, we apply the bound on the serving of requests in the following epoch.

## 3.2 Implemention of Derived Instantaneous Throughput Bounds

The CART model calculates a program's throughput bounds individually for each data server. However, in a shared storage system consisting of multiple servers over which files are striped, the resource needed for maintaining a minimal throughput bound at a server not only depends on the bound, and the spatial locality of requests to the server, but also is affected by the program's requests on other servers and requests of other programs sharing the servers. Therefore, request scheduling for the storage system is concerned not only with implementing throughput bounds for individual programs, but also with the efficiency of the implementation to jointly accommodate multiple concurrent programs with QoS goals.

### 3.2.1 *Coordination of Request Scheduling across Data Servers*

In the implementation of a throughput bound for a program on one server, it is desirable to have a time window in which only requests from this program are served, while requests from other programs, if any, wait for the completion of the window. This is because concurrently serving requests from different programs usually results in frequent costly disk seeks between the data sets accessed by the different programs. However, there is a serious performance challenge in the use of dedicated time windows, namely the potential of long idle times between two consecutive requests from the same program to a data server. If the disk chooses to wait for the next request, as the anticipatory scheduler may do [15], it can be kept idle for too long and make the wait more expensive than seeking to other programs' data. Unfortunately, it is common to have such long idle times in a multi-server system. Because file data is striped, when one server completes serving a request from a program, the next several requests may be for data stored in other servers that may be servicing requests from other programs. This issue of long idle time may be ameliorated to some extent when different processes of the program issue concurrent requests. However, the issue remains as a barrier for efficiently implementing QoS requirements.

To address this issue U-Shape opportunistically coordinates I/O scheduling at different data servers, in the unit of the time window, using a scheme derived from the scheduling mechanism in *IOrchestrator* [34], a strategy for minimizing inter-program I/O access interference in parallel file systems. U-Shape attempts to synchronize the scheduling of requests at different data servers so that during one time window only requests from the same program are served in an effort to improve locality for the served requests. However, this synchronization may increase the idle time between two successive requests from the same program if the request arrival rate for the program is not sufficiently high. To take this into account, U-Shape continuously evaluates the benefit of avoiding frequently disk head seeks against the loss of allowing disk idle to wait for next request. When the benefit is predicted to be greater than the loss, U-Shape uses a dedicated time window to serve requests issued by a single program.

For programs with weak request locality a dedicated scheduling window may never be warranted. All programs that are not selected for dedicated disk service are grouped together to share a common time window. With a group of programs sharing a window, the increased request arrival rate, with consequently more requests enqueued for scheduling, helps disk schedulers work more effectively.

In this way, request scheduling at different servers is synchronized to serve requests in windows of the same program or the same group of programs, one at a time. Because the IOrchestrator scheme does not consider QoS requirements, it determines the sizes of the windows based solely on their achieved throughput to maximize the entire system's throughput. In addition to provisioning of effective resource allocation among concurrently running programs for high system efficiency, U-Shape additionally needs to efficiently implement its QoS-aware throughput bounds on the time-window-based scheduling mechanism.

### 3.2.2 *Determination of Scheduling Window Sizes*

As previously mentioned, there is a daemon at each data server. Any program with a QoS requirement is registered with the daemon. The daemon monitors the access locality of each registered program and runs a trained CART model for each program to derive throughput bounds. In the PVFS2 parallel file system, where U-Shape is prototyped, the daemons send their calculated bounds to a daemon at the file system's metadata server that combines them to obtain the program's net throughput bound. This throughput bound is used by U-Shape to determine the window size.

U-Shape divides wall-clock time into a series of scheduling periods of fixed size $T_{period}$, and each period consists of a number of scheduling windows. U-Shape provides dedicated service in each window of a scheduling period in a round robin fashion. The initial window size $T_{init\_window}$ is inversely proportional to the throughput measured for each window's dedicated service, similar to the method used in IOrchestrator for the highest overall system efficiency. The window sizes $T_{ushape}$ are then adjusted as follows. For each window that is dedicated to one program, we measure the amount of accessed data $D_{ushape}$ in the window. If the throughput of the program in a scheduling period $D_{ushape}/T_{period}$ is smaller than its derived throughput bound $B_{required}$, we extend the window size by $T_{adjust}$, where

$$T_{adjust} = \frac{B_{required} * T_{period} - D_{ushape}}{D_{ushape}/T_{init\_window}},$$

to meet the required throughput. Otherwise, if the program's actual throughput is larger than the derived throughput bound, we reduce the window size by $|T_{adjust}|$.

If the program shares a time window with other programs, we also compare its actual throughput in the period with its

derived throughput bound. If the former is larger, we leave it alone. If it is smaller, a dedicated window will be allocated to the program. The window size is then adjusted as before in the following scheduling period.

If the aggregate window size after the adjustments is larger than the scheduling period, we first reduce or remove windows for best-effort programs, if any. If it is still larger by $T_{overload}$, the window size of any remaining program $k$ is reduced by $T_{cut\_k}$, a portion of $T_{overload}$ proportional to the time remaining until its specified execution deadline. This design takes into account the urgency of programs to meet their deadlines. Because of the reduction on its window size, the program lags behind its deadline by $T_{loss}$, where

$$T_{loss} = D_{ushape} * (T_{cut\_k}/T_{ushape})/B_{required}.$$

This lost time for the program is recorded. In future scheduling periods, if the aggregate window size after U-Shape's adjustments is smaller than the scheduling period by $T_{underload}$, $T_{underload}$ is first used for making up the loss recorded for each program, if any, starting with the most urgent program. If there is still remaining $T_{underload}$, it is evenly distributed to extend the windows of the best-effort programs.

The aforementioned procedure for adapting window size to meet QoS requirements is conducted at the end of every scheduling period so that up-to-date model-derived throughput bounds can be quickly applied in the request scheduling, and the overall system efficiency, as it would be maintained by IOrchestrator, is minimally compromised.

# 4. PERFORMANCE EVALUATION AND ANALYSIS

The performance of U-Shape was evaluated on a system consisting of nine data servers, one also configured as a metadata server for the PVFS2 parallel file system (version 2.8.2), with four compute servers on the client side. All nodes were of identical configuration, each with dual 1.6GHz Pentium processors, 1GB RAM, and a SATA disk (Seagate Barracuda 7200.10, 150GB) with NCQ enabled. Each node ran Linux 2.6.31.3 with CFQ (the default Linux disk scheduler). We used MPICH2-1.2.1 with ROMIO to generate executables of MPI programs. All nodes were interconnected with a switched Gigabit Ethernet network. File data was striped over the data servers using PVFS2's default striping unit of 64KB. To ensure that all data were accessed from disk the system buffer caches of the compute nodes and data servers were flushed prior to each test run.

As mentioned in Section 3, the IOrchestrator scheduling strategy was adapted and augmented to support window-based request scheduling in U-Shape. As in IOrchestrator, there is a daemon at the metaserver to track which files were opened by a program and pass the information it to daemons at data servers so that the data servers can know from which program requests are issued. The daemon in the metadata server computes the current window sizes and informs the daemons at the data servers of the updated sizes for implementing the scheduling. In this work, we set the size of the scheduling period (corresponding to IOrchestrator's scheduling window) to 1000ms and the size of the epoch for evaluating access locality and deriving instantaneous throughput bounds to 100ms. As specification of I/O QoS requirement in terms of constant throughput is the most common in current practice and contemporary research literature [18, 9, 10], in the evaluation we compare U-Shape with the scheme attempting to maintain constant throughputs.

## 4.1 Workloads

In the evaluation of U-Shape we use various workloads, including the synthetic workload used to characterize the storage system and train CART models, two microbenchmarks with distinct and simple access patterns, and two macrobenchmarks adapted from real applications with more complex access behaviors.

### *4.1.1 Synthetic Workloads*

We designed a synthetic workload generator to cover a large spectrum of spatial locality. This MPI-IO program reads/writes a file with varying access patterns. Request sizes range from 4KB to 128KB, the distance between two consecutive requests varies from 0GB and 10GB. Because performance is more sensitive to changes in distance when the distance is small, the distance increments are smaller for smaller distances. The requests were cooperatively sent by eight processes of the program using collective I/O.

### *4.1.2 Microbenchmarks*

The two microbenchmarks are *mpi-io-test* and *ior-mpi-io*. *Mpi-io-test* is an MPI-IO benchmark from the PVFS2 distribution [21]. In our experiments we ran the benchmark with eight MPI processes to read or write one 10GB file. Each process $p_i$ accesses the $(i+8j)^{th}$ 64KB segment at call $j$, for $0 \leq i < 8$, yielding a fully sequential access pattern.

*Ior-mpi-io* is a program in the ASC Purple Benchmark Suite developed at Lawrence Livermore National Laboratory [13]. In this benchmark each of the eight MPI processes is responsible for reading its own 1/8 of a 2GB file. Each process continuously issues sequential requests, each for a 32KB segment. The processes' requests for the data are at the same relative offset in each process's access scope of 256MB. The program's access pattern as presented to the storage system is effectively random.

### *4.1.3 Macrobenchmarks*

The two macrobenchmarks are *BTIO* from the NAS parallel benchmark suite [20] and *S3aSim* from Northwestern University [26].

*BTIO* is a Fortran MPI program designed to solve the 3D compressible Navier-Stokes equations using the MPI-IO library for its on-disk data access. We ran the program using four processes with an input size coded as *A* in the benchmark, which generates a data set of about 419MB, using non-collective I/O operations.

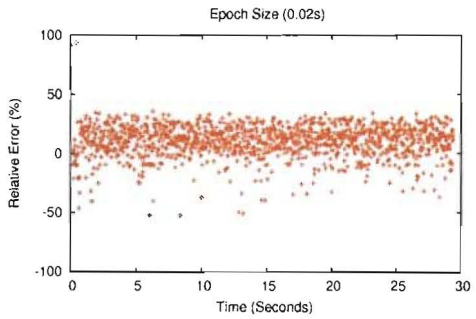
*S3aSim* is a program widely used in computational biology for sequence similarity search. Eight processes were spawned to run the program, which accesses 552MB data during executions.

## 4.2 Accuracy of the CART Model

We evaluate the accuracy of a trained CART model for benchmark *S3aSim*. We first ran the benchmark to collect feature vectors for each epoch and their corresponding throughputs. These statistics were used to train a CART model. This model was then used for estimating the throughputs of epochs of a real run. Different runs of the program may have different access patterns because of different inputs or a different file layout on disk. In *S3aSim* request

sizes vary. In general it is insufficient to rely solely on the benchmark itself for training; the synthetic workload generator is used to collect statistics to train the model with a more comprehensive coverage of access patterns.

We then performed a real run of *S3aSim* on the storage system with 50% of its service time allocated to the program. The trained CART model ran on the data servers monitoring the program's access pattern and throughput $B_{model}$ for each epoch. Independently, we measured the actual throughput $B_{actual}$ in each epoch of the run. The relative error, used to quantify the prediction accuracy, is defined as $(B_{model} - B_{actual})/B_{actual}$. Figure 3(a) shows the relative error of the model predictions for the first 30s execution with 20ms epoch size, and Figure 3(b) with 40ms epoch size. The results show that the errors are roughly equally distributed around zero, and that the errors are reduced when we increase the window size from 20ms to 40ms. When the epoch is 40ms, 80% of relative errors are less than 29%. Increasing the epoch size to 100ms, the default value for U-Shape, 90% of relative errors are less than 14%, which is quite acceptable for our purposes.



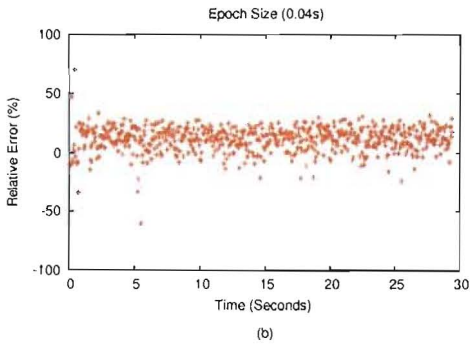




**Figure 3: Relative errors of predicted epoch throughputs with different epoch sizes: (a) Epoch size 20ms; (b) Epoch size 40ms.**

## 4.3 Performance Isolation

An important feature of U-Shape is that it maintains a predetermined resource bound, instead of a fixed throughput bound, in its implementation of QoS requirements. In this way the resource allocated to one program can be well controlled to (1) maintain possibly varying instantaneous throughput bounds derived for meeting the program's QoS goal; and, (2) implement performance isolation among programs by keeping any program from acquiring more resource than planned. To demonstrate the achievement of these objectives, we run *ior-mpi-io* and *mpi-io-test* concurrently. As checkpointing is commonly used to protect a parallel application from failures [3], we use the ill-behaved *ior-mpi-io* to model an interval of execution between two consecutive checkpoints in a long-running application, interposed between two checkpointing operations. For checkpointing we replicate the benchmark's 2GB data files on a dedicated disk partition, for which accesses are fully sequential.

Figure 4(a) shows throughput for each epoch of the benchmarks when each runs with exclusive use the storage system. The execution times for these dedicated runs are 53.5s and 50.5s for *ior-mpi-io* and *mpi-io-test*, respectively. The figure shows two spikes for *ior-mpi-io* corresponding to the sequential access of checkpointing (210MB/s), and a much lower throughput in between due to its random data access (61MB/s), while *mpi-io-test* maintains a high throughput of approximately 211MB/s.

Suppose users recognize that it would take a longer to run their programs on shared storage and accordingly allow the execution times to be doubled. Specifically, the QoS goals for *ior-mpi-io* and *mpi-io-test* are set to 110s and 100s, respectively. To meet these goals the required average throughputs for *ior-mpi-io* and *mpi-io-test* are 56MB/s and 95MB/s, respectively, for a system using average throughput for meeting QoS goals. Figure 4(b) shows the throughput variations when the two programs run concurrently on the system attempting to maintain a fixed average throughput. When *ior-mpi-io* accesses randomly from 37s to 92s but demands a resource allocation for a throughout bound calculated partly with sequential access speed, it acquires disproportionate storage resource and makes *mpi-io-test*'s throughout drop significantly. Consequently neither maintains its respective average throughput and so violate the QoS requirements by 13% and 40%, respectively.

Figure 4(c) shows the derived and actual throughputs in each epoch in the programs' concurrent run on the system managed by U-Shape. By maintaing the derived throughputs, U-Shape enables the two programs to attain their respective resource allocations. When *ior-mpi-io* issues random requests, the CART model detects the pattern and automatically derives a lower bound (reduced from 102MB/s to less than 40MB/s) for the system to maintain, and the system can comfortably meet its new bound without starving *mpi-io-test*. The result is that *mpi-io-test* is little affected by access pattern changes in *ior-mpi-io*. As Figure 4(c) shows, measured throughputs oscillate around the derived ones, sometimes by a large margin such as for *mpi-io-test*. This is because U-Shape uses window-based scheduling. When a window for a program is scheduled, the program receives a throughput higher than required. In the other times, its throughput becomes much lower. As the window size is calculated by U-Shape according to the derived throughout, the program's average throughput in a larger time period is almost the same as the derived one, as shown in the figure. Both programs can proceed at a speed constantly adapting to the current access pattern towards their respective spec-

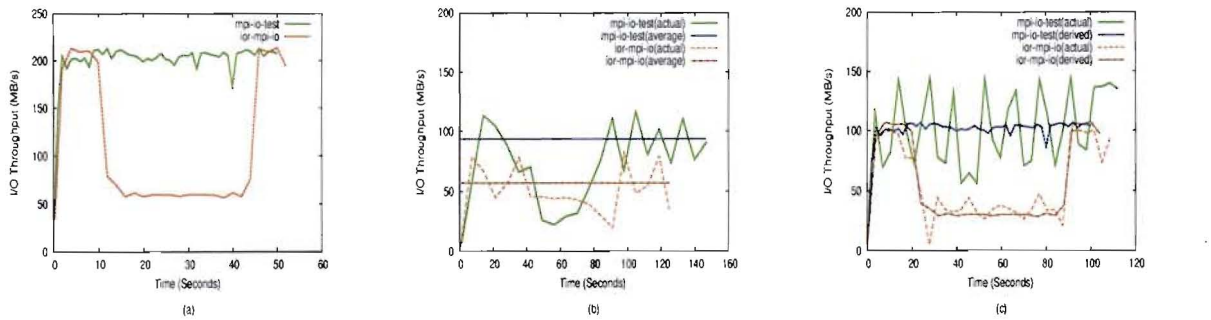



**Figure 4: Throughputs during the execution of benchmarks *mpi-io-test* and *ior-mpi-io*: (a) Measured throughputs when each of the programs runs exclusively on the storage system; (b) Measured throughputs and required average throughputs when the two programs run concurrently on the shared storage system attempting to meet average throughput bounds; (c) Measured throughputs and derived instantaneous throughputs when the two programs run concurrently on the shared storage system using U-Shape.**

ified deadlines. Our measurements show that the execution times of *ior-mpi-io* and *mpi-io-test* are 110s and 105s, respectively, essentially meeting the users' QoS goals.

In the above experiment, with the required QoS goals, or 2X execution times, we show that U-Shape can adhere to the resource allocation planned for the QoS goals. As the required execution times were chosen within an interval in which U-Shape makes a difference in meeting QoS goals, we also experimented with requirements on the execution times outside that interval. When the required times are 1.5X the respective programs' dedicated execution times, as shown in Table 1, both the system using the average throughput and U-Shape are unable to meet the deadlines because of the limited capacity of the storage system. Even so, the execution times of *ior-mpi-io* and *mpi-io-test* with U-Shape are 7% and 31% less than that using average throughput bounds because of bounded resource allocation. When the required times are 3.0X the respective programs' dedicated execution times, both systems can meet the QoS goals. In this experiment also we ran a background program performing data backup to opportunistically absorb surplus I/O capacity.

| QoS | Times (Required) | Times (Average) | Meet? | Times (U-Shape) | Meet? |
|---|---|---|---|---|---|
| 1.5X | 83s/80s | 121s/150s | No | 112s/104s | No |
| 2.0X | 110s/100s | 124s/147s | No | 110s/105s | Yes |
| 3.0X | 160s/150s | 164s/154s | Yes | 162s/156s | Yes |

**Table 1: A comparison of actual execution times of benchmarks *ior-mpi-io* and *mpi-io-test* (shown in each cell in that order) under the scheduling scheme using average throughput and U-Shape with different required execution times. A QoS goal is considered to be met when the execution time is less than 5% larger than the corresponding required time.**

## 4.4 Storage System Efficiency

Another design goal of U-Shape that is as important as QoS assurance is the storage system's efficiency. To evaluate the impact of adaptation of resource partitioning by U-Shape on the efficiency of the storage system, we run multiple instances of *BTIO*. For *BTIO*, I/O times and compute times are interleaved during most of program's execution and the I/O access pattern is random. With dedicated I/O, compute time accounts for approximately 67% of its execution time of 223s. We assume that each *BTIO* instance allows 4.5X execution slowdown, that is, the end-users' required execution time is 1004s. Because the efficiency of data access is largely determined by seek distances, we vary the on-disk distance between each two contiguous files accessed by the different instances. We use distances of 0GB, 10GB, 20GB, and 30GB. To obtain insight into how U-Shape's scheduling policy responds to accesses of differing locality, we keep *mpi-io-test*, which issues sequential requests and demands a constant 80MB/s throughput, running in the background. Table 2 show the numbers of BTIO instances the storage system can accommodate without violating the QoS goals, as a function of file distance, using average throughput and U-Shape to specify QoS goals.

| File Distances | 0GB | 10GB | 20GB | 30GB |
|---|---|---|---|---|
| Average | 2 | 1 | 1 | 1 |
| U-Shape | 4 | 3 | 2 | 2 |

**Table 2: A comparison of the largest number of *BTIO* instances the storage system can accommodate without QoS violation.**

U-Shape can service more instances of *BTIO* than the system with average throughput bounds, a direct consequence of U-Shape's maintaining windows' efficiency. When file distance is 0GB, most of the seek times for disk heads to serve requests from different instances are smaller than the wait times for the arrival of next request from the same instance. The wait time of an instance is determined by its reuse distance, the time gap between two successive requests from the instance to the storage. Since *BTIO*'s requests are interleaved with many small compute times, its reuse distances are large compared with the average seek times, as shown in Figure 5. Accordingly U-Shape groups the *BTIO* instances in one scheduling window and serves them together to allow

the disks to remain busy. However, *mpi-io-test*, which constantly issues sequential requests, warrants a dedicated window, which U-Shape provides. Thus the system resources can be efficiently used, improving the entire storage system's throughput and allowing more instances to achieve their QoS goals. Figure 6 shows the aggregate throughput of four concurrently running *BTIO* instances. The throughput achieved by U-Shape is much higher—by 3–7 times—than that using average throughput bounds. The throughput of *mpi-io-test* is maintained at 80MB/s in both systems since both provide dedicated service windows to it.

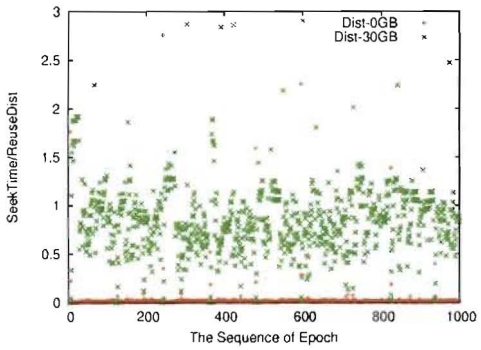



**Figure 5: The ratios of average seek time and reuse distance for each epoch with file distances of 0GB and 30GB for the first 1000 epochs.**

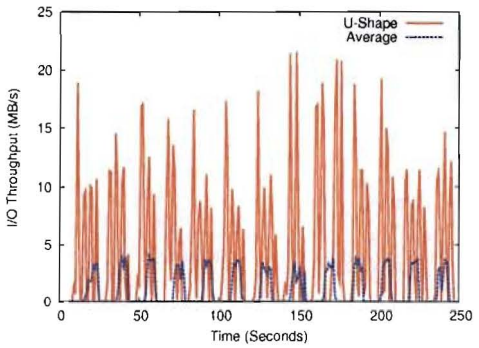



**Figure 6: Aggregate throughput of *BTIO* instances in the system using U-Shape and using average throughput bounds, for the first 250 seconds.**

When we increase the file distance from 0GB to 30GB, the ratios of seek time and reuse distance become much larger because the seek times between files accessed by different *BTIO* instances increase significantly (Figure 5). The overhead of disk seek times outweighs the cost of disk waiting. After evaluating the locality change, U-Shape serves the requests from different instances in different dedicated windows for greater I/O efficiency.

## 4.5 Performance of an Overloaded System

To show how U-Shape can flexibly allocate resources in an overloaded system to meet QoS goals, we run *S3aSim* together with *mpi-io-test*. The execution time of *S3aSim* is 19.8s when it runs exclusively on the storage system. The QoS goal is 79.2s, a 4X slowdown. The QoS goal for *mpi-io-test* is 141s, or an average throughput of 78MB/s. To maintain the 78MB/s throughput for *mpi-io-test*, about 50% of the I/O capacity must be allocated. Running it together with *S3aSim* overloads the system. When the resource shortage is evenly shared by the two programs, the execution time of the *S3aSim* program is 151s, almost double the required QoS goal, and *mpi-io-test*'s execution time is 145s, barely meeting its deadline, as shown in Figure 7(a). Figure 7(b) shows the I/O throughput of the two programs when U-shape is applied. When U-Shape detects the shortage, it forces *mpi-io-test* to have a more greatly reduced service window than *S3aSim* because *mpi-io-test* has a deadline much later than *S3aSim*. Specifically, the ratio of service window times between *S3aSim* and *mpi-io-test* is 3.5 from 0.6s to 86.0s in the execution. The significant reduction of *mpi-io-test*'s throughput makes it possible for *S3aSim* to meet its deadline. The performance losses are recorded for both programs. As soon as *S3aSim* is completed, U-shape allocates all I/O capacity to *mpi-io-test*. By scheduling in this way, *mpi-io-test* misses its deadline by only 9%, *S3aSim* misses its deadline by only 3%, on the overloaded system.

## 4.6 Overhead Analysis

While U-Shape demonstrates clear advantages in various settings, it incurs both offline and online overheads. Specifically, the offline overhead includes the time for profiling runs of programs and for model training. Though the offline overhead may not be visible to end users it is desirable that it be of nominal magnitude. The online overhead includes two major components: 1) the time for deriving instantaneous throughput; and, 2) the time for determining scheduling window size. We evaluate both of these.

### *4.6.1 Offline Overhead*

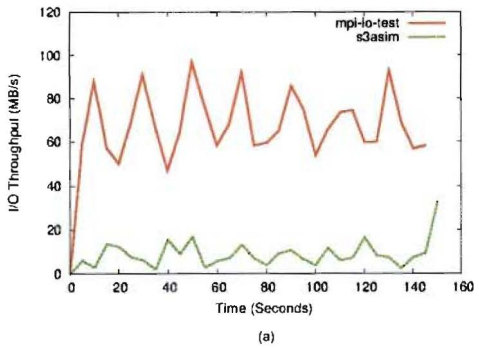
For each benchmark we train a CART model using both a synthetic trace of 20,000 I/O requests, and a randomly selected segment of 10,000 I/O requests from the trace obtained from the profiling run of the benchmark, thus customizing the model to the benchmark while retaining the generality informed by the synthetic trace. Table 3 shows that the training time for each of the four benchmarks is small.

| *mpi-io-test* | *ior-mpi-io* | *btio* | *S3aSim* |
|---|---|---|---|
| 6.6s | 8.0s | 8.4s | 7.6s |

**Table 3: Model training times for the benchmarks.**

### *4.6.2 Online Overhead*

In the investigation of the impact of online overhead we do not directly measure the overhead times because they can be overlapped with real I/O service times or programs' compute times. Instead we measure the increase in the programs' execution times due to U-Shape's overhead. To accomplish this we need to nullify U-Shape's other effects on execution time. We choose *mpi-io-test* as representative, with and without using U-Shape. We set its required execution time equal to



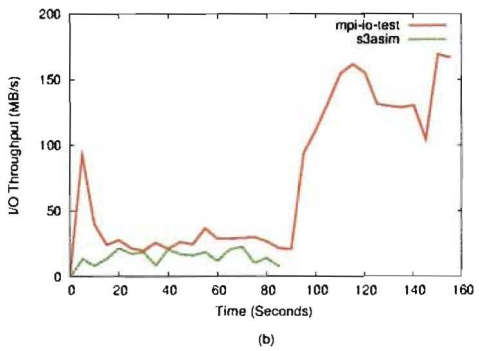




**Figure 7: Throughput of concurrently running *mpi-io-test* and *S3aSim*: (a) Throughput when resource shortage is evenly shared by the two programs; (b) Throughput when U-Shape is used.**

the time of its dedicated run. Since epoch size determines the overhead with the CART model for deriving throughput bounds, we measure the execution time of the program with epoch sizes of 50ms, 100ms, and 150ms. As shown in Figure 8, the online overhead of U-Shape accounts for less than 5% of the program execution time with a 50ms epoch. With larger epoch sizes, the overhead becomes smaller. To make a tradeoff between the overhead and responsiveness in the derivation of throughput bound, we use a 100ms epoch in practice. Note that these measured overheads are exposed in the dedicated runs of the programs with U-Shape. In practice U-Shape could be turned off in this scenario.

# 5. RELATED WORK

Sharing of a cluster of data servers in a high-performance computing installation is common practice, and storage is also increasingly consolidated in other high-end and high-capacity computing environments for economies of scale in system, maintenance, and energy costs. While shared storage service provides significant benefits, QoS assurance for its users is one of the more critical issues to be addressed and much research work has been focusing on it, including QoS specification, characterization of storage system performance, and QoS-aware resource partitioning and scheduling.

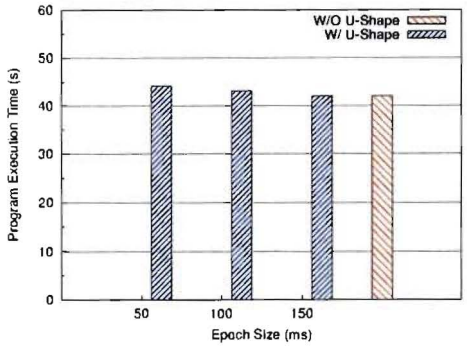



**Figure 8: Execution times of *mpi-io-test* with and without U-Shape overhead for different epoch sizes.**

## 5.1 Performance Interfaces

The commonly used metrics for users to present their QoS requirements to storage services are throughput and response time of data access [11, 8]. A serious problem with the use of these simplistic parameters is that service quality can be significantly affected by the characteristics of I/O workloads. To address this, researchers have developed I/O workload models to derive performance bounds [6, 27, 29, 30]. Because a workload's characteristics can vary widely, constraints have been introduced to guarantee QoS only when the constraints are met. For example, in pClock scheduling the response time of a request is bounded only if the request burstiness (the number of pending requests) and request arrival rate are less than pre-set thresholds [8]. However, the relationship between the imposed constraints and the guaranteed performance may not reflect the performance expectation from users. To provide flexibility, Facade, a high-end storage system prototype, allows increased response times when I/O request arrival rate increases [18]. However, it is a challenge for users to determine a series of performance bounds, each associated with one I/O rate.

Spatial locality is a unique property of the storage system and even harder than I/O rate to encode in a performance interface. As such, performance bounds are specified without regard to sequential and random access in most systems [2, 5, 8, 9, 10, 11, 16, 18, 35]. In addition, applications may be required to explicitly indicate their I/O access pattern [24]. An exception is the Argon storage server in which a client requests a fixed fraction of a server's capacity, which is equivalent to setting performance bounds that are aware of spatial locality [32]. This, however, requires users to know the server's total capacity and its relation to application performance. In contrast, U-Shape frees users from the burden of specifying their QoS requirements with specific I/O behaviors and I/O performance goals, or having specific knowledge of server capacity: the machine learning technique automatically derives I/O performance goals.

## 5.2 Characterization of Storage System Performance

For estimating the required throughput for various access patterns, storage systems' performance behaviors must be

well characterized. For this purpose there are three candidate methods, namely, analytic system modeling, simulation, and black-box modeling. Because of their nonlinear, state-dependent behavior, building accurate analytic models for disk drives is a non-trivial task [25, 33]. For a storage system consisting of multiple data servers, each with one or more disks, this modeling method seems infeasible.

Construction of a disk simulator modeling performance-relevant components of a disk, such as device drivers, buses, controllers, and adapters, is another method for performance estimation [7]. However, it requires human expertise on the targeted device or system, which may include proprietary configurations and use of algorithms and optimizations that are not disclosed.

In contrast, the so-called *black box* method can serve as a more general-purpose approach. It treats the system as a black box without assuming the knowledge of its internal components or algorithms. In this approach the training data set, containing the quantified description of characteristics of input requests and their corresponding responses from the system, is recorded [1, 22], and fed into a statistical model [17] or a machine learning model [31, 19]. To predict a response to an input, some form of interpolation is required. It is recognized that the accuracy of the method relies on the appropriate selection of training set data and the design of feature vectors (the set of input characteristics) [17, 31]. In U-Shape we use the machine-learning method to model the shared storage system by selecting relevant training data. In addition, the effectiveness of the model in U-Shape relies only on an aggregate performance metric, or the throughput in a time epoch, rather than latencies of individual requests. This makes the method especially useful in our work.

### 5.3 QoS-Aware Resource Partitioning

Many QoS-aware resource partitioning policies enforce disk bandwidth isolation and guarantee I/O service quality by tagging requests from different request streams with deadlines (or finish times) calculated from users-specified performance bounds and estimated service times [11, 23]. While service time is heavily dependent on spatial locality in disk-based storage systems, the locality is usually not included in the performance interface and random access is usually assumed. However, this can cause resource over-provisioning. To fix this problem, Stonehenge allows additional streams to keep joining the system until the system is found to be overloaded [11]. They are forced to use this trial-and-error method because the performance interface does not contain information on spatial locality and planning of resource provisioning is difficult. In contrast, applications' resource consumption is implicitly contained in the model-derived I/O performance interface in the U-Shape scheme to enable well-planned scheduling. The resource consumption is also contained in Argon by setting explicit quota of disk service time for each stream [32]. However, in a shared environment it is a challenge to know to which stream a service time should be attributed [23]. In the implementation of derived throughputs, U-Shape leverages the mechanism proposed by IOrchestrator [34], which enables efficient I/O resource allocation in a storage system consisting of multiple data servers.

## 6. CONCLUSIONS

We have proposed a scheme, U-Shape, to support end-user-specified QoS requirements in the form of execution times for programs using shared storage systems. By not requiring QoS requirements in the form of throughput and latency bounds, U-Shape provides a highly convenient performance interface and automatically derives corresponding I/O requirements. By using a machine learning technique, U-Shape can ensure that a program receives sufficient service to meet its QoS requirements with bounded resource demands. As such U-Shape does not need to explicitly consider the dynamics of the service of I/O requests, such as spatial locality and arrival rate, that are difficult to accurately quantify. Our experimental evaluation using representative benchmarks and scenarios shows that U-Shape can faithfully and efficiently meet realistic QoS requirements specified by end users, and provides strong performance isolation. U-Shape maintains high overall system efficiency even when the system is overloaded, and imposes minimal run-time overhead.

## 7. REFERENCES


[1] E. Anderson. Simple table-based modeling of storage devices. *Technical Report HPL-2001-04*, HP Laboratories, 2001.

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

[3] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: a checkpoint file system for parallel applications. In *Proc. of the Supercomputing Conference*, 2009.

[4] S. Barua, R. Thulasiram, and P. Thulasiraman. High Performance Computing for a Financial Application Using Fast Fourier Transform. In *Proc. of EURO-PAR*, 2005.

[5] D. Chambliss, G. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. Lee. Performance virtualization for large-scale storage systems. In *Proc. of the Symposium on Reliable Distributed systems*, 2003.

[6] M. Calzarossa and G. Serazzi. Workload characterization. In *Proc. of the IEEE*, 81(8), 1993.

[7] The DiskSim simulation environment (version 3.0). *http://www.pdl.cmu.edu/DiskSim/*, online document, 2009.

[8] A. Gulati, A. Merchant, and P. Varman. pClock: an arrival curve based approach for QoS guarantees in shared systems. In *Proc. of the ACM SIGMETRICS Conference*, 2007.

[9] A. Gulati, I. Ahmad, C. A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proc. of the 8th USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[10] A. Gulati, A. Merchant, and P. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proc. of the 9th USENIX Symposium on Operating System Design and Implementation*, 2010.

[11] L. Huang, G. Peng, and T. Chiueh. Multi-Dimensional Storage Virtualization. In *Proc. of the ACM*

*SIGMETRICS Conference*, 2004.

[12] Geerd-R. Hoffmann. High performance computing and networking for numerical weather prediction. In Proc. of the International Conference and Exhibition on High-Performance Computing and Networking, 1994.

[13] Interleaved or Random (IOR) benchmarks. *http://www.cs.dartmouth.edu/ pario/examples.html*, online document, 2010.

[14] IBM Deep Blue, *http://www.research.ibm.com/deepblue.*

[15] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proc. 18th ACM Symposium on Operating Systems Principles*, 2001.

[16] W. Jin, J. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proc. of the ACM SIGMETRICS Conference*, 2007.

[17] T. Kelly, I. Cohen, M. Goldszmidt, and K. Keeton. Inducing models of black-box storage arrays. *Technical Report HPL-2004-108*, HP Laboratories, 2004.

[18] C. Lumb, A. Merchant, and G. Alvarezg. Facade: virtual storage devices with performance guaratees. In *Proc. of the USENIX Annual Technical Conference*, 2003.

[19] M. Mesnier, M. Wachs, R. Sambasivan, A. Zheng, and G. Ganger. Modeling the relative fitness of storage. In *Proc. of the ACM SIGMETRICS Conference*, 2007.

[20] NAS Parallel Benchmarks, NASA AMES Research Center, *http://www.nas.nasa.gov/Software/NPB*, online document, 2009.

[21] PVFS, *http://www.pvfs.org*, online document, 2010.

[22] F. Popovici, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Robust, portable i/o scheduling with the disk mimic. In *Proc. of the 2003 USENIX Annual Technical Conference*, 2003.

[23] G. Peng and T. Chiueh. Availability and fairness support for storage QoS guarantee. In *Proc. of the IEEE ICDCS Conference*, 2008.

[24] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. Wong, C. Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. In *Proc. of the The European Conference on Computer Systems*, 2008.

[25] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3), 1994.

[26] S3aSim I/O Benchmark, *http://www-unix.mcs.anl.gov/thakur/s3asim.html*, online document, 2009.

[27] E. Smirni and D. Reed. Workload characterization of input/output intensive parallel applications. In *Proc. of the 9th International Conference on Computer Performance Evaluation: Modeling and Techniques and Tools*, 1997.

[28] The Human Genome Project, *http://www.ornl.gov/hgmis/.*

[29] N. Tran and D. Reed. Arima time series modeling and forecasting for adaptive I/O prefetching. In *Proc. of the 15th international conference on Supercomputing*, 2001.

[30] S. Uttamchandani, L. Yin, G. Alvarez, J. Palmer, and G. Agha. Chamelon: a self-evolving, fully-adaptive resource arbitrator for storage systems.. In *Proc. of the USENIX Annual Technical Conference*, 2005.

[31] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. Ganger. Storage device performance prediction with cart models. In *Proc. of the 12th MASCOTS Conference*, 2004.

[32] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. Ganger. Argon: performance insulation for shared storage servers. In *Proc. of the 6th USENIX Conference on File and Storage Technologies*, 2007.

[33] B. Worthington, G. Ganger, and Y. Patt. Scheduling algorithm for modern disk drives. In *Proc. of the ACM SIGMETRICS Conference*, 1994.

[34] X. Zhang, K. Davis, and S. Jiang. IOrchestrator: improving the performance of multi-node I/O Systems via inter-server coordination. In *Proc. of the ACM/IEEE Supercomputing Conference*, 2010.

[35] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. *ACM Transanction on Storage*, Vol. 2, Issue 3, 2006.

[36] Y. Zhang, E. Kung, and D. Haworth. A PDF Method for Multidimensional Modeling of HCCI Engine Combustion: Effects of Turbulence/Chemistry Interactions on Ignition Timing and Emissions. In *Proc. of the 30th International Symposium on Combustion*, Vol. 30, Issue 2, 2004.