A QoS-aware Scheduling Scheme for Software-Defined Storage Oriented iSCSI Target

Xianghui Meng\textsuperscript{1,2}, Xuewen Zeng\textsuperscript{1}, Xiao Chen\textsuperscript{1}, Xiaozhou Ye\textsuperscript{1,*
\textsuperscript{1}National Network New Media Engineering Research Center, Institute of Acoustics Chinese Academy of Sciences, Beijing 100190, P.R. China
\textsuperscript{2}University of Chinese Academy of Sciences, Beijing 100190, P.R. China
*Corresponding author(E-mail: yezx@dsp.ac.cn)

Abstract
Software-defined Storage (SDS) which uses virtualization and pooling technologies provides a new and flexible way to deploy a shared Storage Area Network (SAN). However, sharing of disk resources can cause performance degradations in an SDS virtualization environment. Concurrent I/O applications which have diverse latency requirements and send request to different logic units (LUNs) co-located on the same shared storage target will lead to unpredictable performance for the SDS. This paper focuses on providing differentiated service to different applications and ensuring Quality of Service (QoS) of time-critical applications in the iSCSI SAN of SDS. An analysis model is presented to describe this problem. Based on this analysis model, a QoS-aware scheduling framework for the SDS oriented iSCSI target (iSDS-QoS) incorporated with iSCSI storage controller is proposed. The iSDS-QoS employs a priority-based strategy to ensure a predictable latency performance of critical applications. iSDS-QoS consists of two components – latency estimation and priority adjustment. It schedules I/O requests by dynamically adjusting their priorities based on estimated latencies, thus making latency sensitive applications achieve their required deadlines and guarantee their QoS. Experiments on real-world I/O workloads demonstrate that, it can improve the achievement rate of required deadlines from 36.7% to 83.3%.

Key words: Software-defined storage, quality of service, network storage, scheduling algorithm, I/O performance

1. INTRODUCTION

In the era of cloud computing and big data, there has been an increase in the amount of network data creating new challenges for storage systems [1]. Software-defined storage (SDS) [2] typically includes a form of storage virtualization that decouples the physical storage hardware from the software that manages it. SDS has the potential to provide shared storage to clients through storage area network (SAN) – this can solve the capacity, availability and manageability problems related to storage. However, inherent sharing of storage resources can lead to performance degradations. Multiple applications requesting different logic unit numbers (LUNs) located on the same iSCSI [3] server will contend for the same disk resource. This contention for disk resource depends on a number of factors - the type of storage, configuration of the LUNs and the number of LUNs co-located on the same iSCSI target - which may result in unpredictable delays and performance loss. I/O requests which have been optimized on different iSCSI initiators can be out of order as they reach the iSCSI target server concurrently, leading to increased latency [4]. The holistic performance of all I/O operations will decrease when the number of LUNs working concurrently on the target increases. This is because concurrent I/O of different LUNs contend with each other for disk access leading to interference at the disk level. Resolving such issues is pertinent for applications to achieve their service level objectives (SLOs).

A major concern of shared storage systems is the reduced system efficiency due to the interference of different workloads with different performance requirements in terms of latency and throughput. A prerequisite for efficient sharing is that concurrent applications are isolated from each other, so that interactive or time-critical applications such as online video streaming applications and transaction-based applications are not affected by other I/O intensive workloads [5]. For latency sensitive I/O applications, undesirable delays may occur if all concurrent I/O applications are allocated equal resources regardless of their I/O requirements. Providing differentiated services to different I/O workloads is very necessary. Many previous works [14-17] have studied the problem of QoS in SDS environment, but little work has been done on iSCSI SAN, whose performance improvements can lead to further promotion of the whole SDS. It is significantly more challenging to provide performance isolation and QoS guarantees among various applications [14], and we are interested in enabling quantitative QoS guarantees for various types of time-critical applications.

In this paper, we present an I/O scheduling scheme iSDS-QoS (iSCSI-based SDS oriented QoS scheduling). This scheme provides differentiated services to different I/O applications requesting multiple LUNs...
co-located on the same shared storage based on their latency requirement. For the iSCSI storage stack, block-level schedulers are severely limited by their inability to gather information from, and exert control over other levels [6]. Therefore, we add a QoS-aware scheduling scheme into the iSCSI target controller above the block-level scheduler in cooperation with it. The main contributions of this work are summarized as follows:

- We propose an analysis model to demonstrate that concurrent I/O applications requesting multiple LUNs co-located on the same shared storage target in an SDS environment will lead to performance degradation.
- We propose a dynamic scheduling scheme iSDS-QoS which is QoS-aware and provides differentiated services to different applications to ensure QoS.
- We prove through benchmarking tools and real-world I/O workloads that the proposed method can make latency sensitive applications reach the required deadlines and guarantee their QoS.

2. RELATED WORKS

There is a large body of research [9-20] on providing differentiated services among different I/O applications on shared storage systems. These studies, however, have been mostly based on classical operating systems and have rarely considered iSCSI SAN in SDS environments. SCAN [7], an elevator algorithm, widely used in Linux, does not consider time constraints in the scheduling process. Earliest Deadline First (EDF) [8] is usually used in hard real-time environments where I/O requests have completion deadlines. However, EDF has a heavy overhead in terms of magnetic head seeking time and disk rotational latency, leading to poor disk utilization. The QoS concept was formally proposed by Christopher R. Lumb [9] for the first time in the storage field in Façade. Façade is a virtual storage controller which sits between hosts and storage devices in the network and throttles individual I/O requests from multiple clients [9]. Previous studies [10-13] have used different scheduling methods to alleviate I/O contention and guarantee specific QoS in shared storage environment. However, only a limited number of the aforementioned studies have focused on policies in the data plane. A data plane is a very large bucket that provides different types of storages such as NAS, object, and block storage. The impact of a data plane on the overall SDS is critical. The present study differs significantly from these studies since this work is focused on the data plane, specifically the iSCSI SAN.

Previous studies [14-16] provided QoS in multi-tiered storage systems with solid state drive (SSD) and traditional hard disk drive (HDD) to build hierarchical storage. Typically, IOFlow [17] designed a SDS architecture and borrowed several SDN ideas and applied them to the shared storage concept. However, it focused only on I/O requests from Virtual Machines (VMs) to the storage. Studies [15, 18, 19] provided differentiated I/O services to multiple applications in the VM hypervisor. Qiwen Zha designed an I/O scheduling algorithm for a soft real-time service oriented iSCSI storage system in the iSCSI initiator [20]. While the above works have achieved specific QoS based on a VM hypervisor or client side, limited attention has been ascribed to the iSCSI target controller.

There are two major differences between the present study and previous studies: a) the present study achieves QoS of I/O applications in iSCSI storage controller as opposed to previous studies which use iSCSI initiators or VM hypervisor b) scheduling on LUNs for the current study is fine-grain as opposed to the course-grained scheduling on VMs.

3. ANALYSIS MODEL AND SOFTWARE ARCHITECTURE

3.1 Analysis Model

An I/O request R is represented by the attribute set R = (D, S, P, L, W). D represents the relative deadline; S is the total requested data size; P stands for the priority of R; L indicates the estimated or actual latency of R; and W is the disk bandwidth allocated to R. Some applications such as soft real-time services do not have determined deadlines. In such cases, D can be represented by a fuzzy interval [d, D]. The QoS of the completion time of a request can be represented by a membership function of fuzzy deadlines, as shown in Fig.1. If an I/O application finishes within the lower bound of the interval (d), it will achieve an optimal QoS (the optimal value is set to 1). If the actual latency exceeds d, this request will not be cancelled immediately as long as the maximum latency stays within D - however the QoS will degrade with this increase in latency. If the latency exceeds D, the QoS will be set to 0. In this paper, the QoS is defined as Eq. 1 which will be used as a metric later.

\[
QoS = \begin{cases} 
1, & (t < d) \\
\frac{1}{d-D}t + \frac{D}{D-d}, & (d \leq t \leq D) \\
0, & (t > D)
\end{cases}
\]  

(1)
We discuss the parallel scheduling model for N concurrent I/O requests, with the assumption that each LUN is targeted by a single request. The disk service time of a request with data size \( s \) is expressed as

\[
T = T_{\text{tran}} + \tau,
\]

where \( T_{\text{tran}} \) is the transfer time proportional to \( s \), and \( \tau \) is the overhead (includes the seeking time and rotation time), which is a fixed value for each transfer process. Assuming \( n \) concurrent requests have sizes \( s_1, s_2, \ldots, s_n \), respectively, the overall storage device bandwidth is written as \( W \). Each request is accessing a dedicated LUN, thus there are also \( n \) LUNs. In parallel scheduling, if all concurrent requests have the same priority, then each one will occupy \( W/n \) bandwidth on an average. When we take different priorities \( (p_1, p_2, \ldots, p_n) \) into consideration, a request with a priority \( p \) will occupy \( \frac{W}{p} \). The response time of first request can be written as:

\[
T_1 = \frac{n \cdot s_1}{p_1 \cdot W} + \tau.
\]  

(2)

When \( i < 1 \) requests have been completed, the average bandwidth for rest of the requests will be \( W/(n-i+1) \), \( i > 1 \). Therefore, the response time of the \( i \)th request will be:

\[
T_i = T_{i-1} + \frac{(s_i - s_{i-1}) \cdot (n+1-i)}{p_i \cdot W} 
\]  

(3)

where \( 1 < i \leq n \).

We can further expand this to write:

\[
T_i = \sum_{k=2}^{i-2} \frac{(s_k - s_{k-1}) \cdot (n+1-k)}{p_k \cdot W} + \frac{n \cdot s_i}{p_i \cdot W} + \tau
\]  

(4)

\[
T_i = \frac{1}{W} \sum_{k=1}^{i-1} \frac{s_k}{p_k} + \frac{1}{p_i \cdot W} \cdot (n+1-i) \cdot s_i + \tau
\]

The mean response time can be described as:

\[
E_p = \frac{1}{n \cdot W} \sum_{i=1}^{n} \frac{(2n-2i+1) \cdot s_i}{p_i} + \tau, \quad n \geq 2.
\]  

(5)

If all concurrent applications have the same priority, then we can write \( p_i = 1 \). Eq. 4 indicates that a request with a higher priority \( p \) (a larger value of \( p \) represents a higher priority) will have a lower response time \( T \). Therefore, in order to satisfy the requirements to meet the deadline, a time-critical application should be assigned a high priority. From Eq. 5, we can observe that the average response time of all applications is proportional to the number of LUNs – we can validate this from Fig. 2(c).
To analyze the effect of concurrent I/O under an SDS environment, this research examines the variations in the accumulated disk performance with a diverse number of LUNs co-located on the same target. As shown in Fig. 2, the values of the holistic IOPS (Fig. 2a) and throughput (Fig. 2b) for all I/O operations decrease, while the average response time (Fig. 2c) increases with an increase in the number of concurrently working LUNs.

3.2 Software Architecture

In this section, we present the design architecture of the iSDS-QoS in the SDS framework. We describe the features of the SDS framework with QoS enabled. We focus on an application scenario where concurrent I/O applications with varied latency requirements run on multiple LUNs co-located on a single shared storage. The relative layout of the iSDS-QoS with respect to the overall SDS framework is shown in Fig. 3. The iSDS-QoS functions as a module of the iSCSI controller which manages the storage resources on the iSCSI server. The iSDS-QoS is composed of a Latency Estimation module and a Priority Adjustment module whose functions are shown in detail in Fig. 4.
The iSCSI Enterprise Target (IET) [23] is an open source iSCSI target controller software, which includes the user space and the kernel space. Most data processing works are performed in the kernel space. The kernel space contains two main components: the NTHREAD and the WTHREAD. The NTHREAD module is responsible for receiving I/O requests from clients and passing the iSCSI PDU to the WTHREAD. The WTHREAD module is responsible for the read/write processes of the iSCSI data and for building up the block requests. Hence, the most appropriate place to integrate the iSDS-QoS is into the WTHREAD.

The main goals of iSDS-QoS are a) to guarantee prioritized services to time critical applications b) to ensure admissible services for applications without strict latency requirements. This is achieved by assigning appropriate priorities for disk access of I/O applications according to their latency requirements. The attributes of an application, such as the data size and the deadline, can be specified by the user. It is assumed that these attributes remain unchanged throughout the period of execution. Deadlines and sizes are extracted from the iSCSI PDU and then passed to iSDS-QoS as input. The Latency Estimation module first predicts the latencies of requests based on the data sizes and system state and then passes the latency values to the Priority Adjustment module. The Priority Adjustment module is responsible for deciding the priority values based on the requirements of the deadline and the predicted latency values. Finally, the priority values are submitted to the underlying block scheduler for scheduling the disk I/O. The aforementioned process is performed at every scheduling cycle which can be modified as needed based on the access pattern of the application. In our work, we use a scheduling interval of 3 seconds which is the best empirical value on our platform. We use a feedback-based strategy to adaptively adjust the priority of the application according to allocated disk bandwidth on the system and re-allocate the bandwidth based on adjusted priority at every interval. An advantage of our approach is that the priorities are not static and can be dynamically updated according to the state of the system. Moreover, our approach does not need to modify the operating system and can be easily integrated into the iSCSI controller as an extension to support the QoS.
4. DESIGN AND IMPLEMENTATION

The iSDS-QoS mainly consists of two modules – the Latency Estimation module and the Priority Adjustment module. The design architecture is illustrated in Fig. 4. The attributes of the I/O applications (deadlines and data sizes) are provided as inputs to the iSDS-QoS and then it generates the priority values of different applications as the output. The output will be submitted to the block scheduler to be taken into consideration for disk scheduling. The detailed design is described below.

4.1 Latency Estimation

To provide differentiated services for all applications, we need to compare the requirements specified by the applications with their actual performance. In order to ensure that a running application meets its required deadline, it is necessary to obtain a clear estimate of the application’s completion time to decide whether the application needs higher priority service. The latency estimation task performs this based on the current disk bandwidth allocated for an application. It takes the total size of each I/O application as input, also it obtains the bandwidth and the size of the remaining data to be used in the next scheduling cycle. The present work uses the Linux iotop utility to collect information regarding the disk I/O (such as disk read/write bandwidth and priority) for each running application. For each scheduling period, iotop is invoked to record these disk utilization statistics. If one application finishes, the rest of the applications will obtain longer disk access times and hence higher bandwidth, leading to a decrease in the predicted values of latency. At the start of each scheduling cycle, the predicted latency values are computed and provided to the Priority Adjustment module.

4.2 Priority Adjustment

Based on the requirements of the I/O application, the Priority Adjustment module can enable service differentiation for each of the application. For each scheduling period, after obtaining the estimated values of latency from the Latency Estimation module, the Priority Adjustment module computes the estimated latency and I/O requirement of each application to decide whether its required deadline can be met for the current state. If the Priority Adjustment module decides that a critical application's latency requirement cannot be met and the expected deadline may be missed, then this module will dynamically adjust the priorities of the active applications so that the allocated bandwidth for the critical application can satisfy its required performance.

The Priority Adjustment module then will submit the assigned priority values to the underlying block scheduler which will execute the actual disk scheduling and allocate the disk access time accordingly. During the service time of an application, it has exclusive disk access. The higher the priority an application has, the longer is its disk access time, leading to a higher disk bandwidth. Therefore, the assigned priority of an I/O application in turn affects its allocated bandwidth in next scheduling period. After adjusting the priorities, new bandwidths for the next scheduling period will be allocated from among all the I/O applications and be passed back to the Latency Estimation module as feedback for re-computation of the latencies. During every scheduling period, the iSDS-QoS will repeat the above procedure once. Within the system bounds and hardware limitations, the Priority Adjustment module provides a locally optimal solution for disk resource allocation along with a QoS guarantee. It not only provides a QoS to critical applications, but also guarantees acceptable services to noncritical ones. Sometimes it may happen that the Priority Adjustment detects a critical application for which it cannot satisfy the required deadline under the current system states even if the highest priority is assigned to it. In such cases the priority of this application will not be elevated – this prevents the degradation of the performance of other applications.

4.3 Implementation

The block scheduler has no knowledge regarding the I/O characteristics of an application except the size and location of its requests. Under such circumstances, the proposed iSDS-QoS scheme figures out the expected priority values and submits this information to the underlying block scheduler. In order to achieve prioritized services, this work implements the Completely Fair Queuing (CFQ) [21] scheduling algorithm which is commonly used in the Linux kernel. Each application is a process running on the host. The purpose of the CFQ algorithm is to provide fairness to all processes in terms of disk bandwidth. The CFQ algorithm uses a series of per-process queues to group synchronous requests and then allocates time slices for each of the queues to access the disk. The length of the time slice and the number of requests a queue can submit depends on the I/O priority of the given process. CFQ has three priority levels: idle (low priority), best-effort (medium priority) and real-time (high priority). CFQ assigns the longest time slices to real-time processes, while idle processes receive the shortest. A process receives exclusive disk access during its time slice. Therefore, a process with a higher priority naturally receives a higher disk I/O bandwidth. A new process will be assigned a default priority of best-effort, but we can modify this using the ionice command in Linux.
Table 1 Terminology for isds-qos algorithm

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>R=&lt;R1, R2...RN&gt;</td>
</tr>
<tr>
<td>Deadline</td>
<td>D=&lt;D1, D2...DN&gt;</td>
</tr>
<tr>
<td>Total Data Size</td>
<td>S=&lt;S1, S2...SN&gt;</td>
</tr>
<tr>
<td>Disk Bandwidth</td>
<td>W=&lt;W1, W2...WN&gt;</td>
</tr>
<tr>
<td>Latency</td>
<td>L=&lt;L1, L2...LN&gt;</td>
</tr>
<tr>
<td>Priority</td>
<td>P=&lt;P1, P2...PN&gt;</td>
</tr>
<tr>
<td>Time Elapsed</td>
<td>T=&lt;T1, T2...TN&gt;</td>
</tr>
</tbody>
</table>

The pseudo code of the iSDS-QoS scheduling algorithm is shown in Algorithm 1. Notations in the pseudo code are listed in Table I. The priority of a real-time process is the highest, and that of an idle process is the lowest, and a best-effort process has default priority. The scheduling algorithm will execute at every scheduling period, and all the active I/O processes are considered for scheduling. The inputs to the algorithm are the deadlines and total data sizes of all applications; and the output is the computed priority. Our algorithm uses a feedback mechanism, namely, on one hand, it computes the priorities according to the current bandwidth of the application during every scheduling period; on the other hand, the priorities assigned in previous periods also impact the bandwidths.

The Algorithm 1 is divided into two parts: a) The first part is the Latency Estimation whose responsibility is to calculate the predicted value of the latency of each active process according to the current bandwidth W and ProcessedData (which can be obtained using iotop). For a new process, the initial value of ProcessedData is set to 0 and its default priority is set to be best-effort. b) The second part is the Priority Adjustment which is responsible for assigning suitable priority values to the active processes. First it compares the predicted latency with the expected deadline of each process and tries to identify a critical process which is likely to miss the deadline. If such a process exists, it will receive a preferential treatment. If there is no such process, it indicates that all the processes are likely to achieve their deadlines, and hence there is no need to adjust the priorities. For each scheduling period, the iSDS-QoS selects a single critical process (if any). If more than one processes are likely to miss their deadlines then the one (Ri) with the earliest deadline will be selected (Line 10). Next, it finds potential victims of Ri - a victim process Rj is one which has a later deadline than that of Ri and is likely to achieve its required deadline (Line 12). If there are more than one such processes Rj, the module will choose the most suitable victim whose priority is higher than the lowest and whose available execution time before the deadline is the longest (Line 13-14). The priority of such a victim process Rj will be decreased to make way for
the critical process \( R_i \) for bandwidth allocation (Line 15-16). If such a process \( R_j \) does not exist and the current priority of \( R_i \) is not the highest (Line 18), then the priority of \( R_i \) will be incremented (Line 19) to enable the critical process receive more bandwidth. If there is no suitable victim process and priority of \( R_i \) is already at the highest (Line 20), then the \( P_i \) will be adjusted to be the lowest. This is because it is impossible for the critical process to meet its deadline given the current state of the system (it already has the highest priority), we must guarantee that other processes won’t suffer from performance loss.

5. EXPERIMENTS AND ANALYSIS

5.1 Algorithmic Validity

To test the validity of the algorithm we test a scenario where two concurrent running applications are targeted on different LUNs on a single iSCSI target. Both of these processes perform simple random writes. We use the IOMeter to create the two workloads. The only difference between the applications is that the first process A is time sensitive while the other process B is not. Initially, both their priorities are set to best-effort by default. We first give the test values of latency under default case where both applications use default priority. Under the best case, the time critical application is assigned the highest priority while the other application is assigned the lowest priority. From Table 2 we can observe that there is a significant decrease in the latency of application A in the best case (which is also the lower bound of time critical application A) compared to the default case, while the latency of B has only slight increase in the best case of A.

<table>
<thead>
<tr>
<th>Table 2 Values of latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency(seconds)</td>
</tr>
<tr>
<td>Default case</td>
</tr>
<tr>
<td>Best case</td>
</tr>
</tbody>
</table>

Based on the information in Table II, we conduct 10 experiments with different values of the deadline for A.

From Fig. 5 we can see that for experiments 1-3, the deadlines for the application A are greater than the latency of the default case. Therefore both the applications can easily satisfy the deadline requirements without any priority adjustment. In experiments 4-6, the deadlines are earlier than the default latency but still greater than the best latency, so the iSDS-QoS adjusts the priority of B to the lowest in order to reduce the latency of A. Hence, the actual latency of A will meet its deadline requirement. Besides, for B there is only a slight increase in the latency. For experiments 7-10, the deadlines are less than the latency of the best case, which means that its deadlines cannot be met in any case (due to current system limitation). In this case the iSDS-QoS adjusts the priority of A to the lowest which makes its latency exceed the required deadline. From the above experiments we can verify that the iSDS-QoS can make latency sensitive applications achieve their required deadlines and can guarantee their quality of service (QoS) within the system capacity.

5.2 Quality of Service (QoS)
This paper focuses on the QoS of different I/O requests. Hence we can use the overall QoS of all the requests as a key performance indicator. The overall QoS can be deduced from Eq. 1, and can be written as:

\[ QoS_{\text{overall}} = \sum_{i=1}^{n} QoS_i, \quad (6) \]

where \( n \) is the number of concurrent requests. The overall QoS reflects the real-time performance. For comparison, we also conduct experiments using classic algorithms such as a) EDF which has the characteristics of optimal deadline satisfaction and b) SCAN which is an elevator algorithm and has an optimal throughput (we will discuss throughput performance in next subsection). The result of the overall QoS is shown in Fig. 6. EDF performs best in terms of overall QoS because it is a hard-real-time scheduler and only considers the required deadline of requests. Therefore it can achieve more than 90% of all deadlines. Overall QoS of SCAN is the worst due to its lack of consideration for latency – it only considers the seek address order. The iSDS-QoS performs much better than SCAN with an increase of about 45% in performance, because iSDS-QoS considers the latency requirement and adjusts the priority of a latency-sensitive application to meet its deadline. We also observe that the difference between iSDS-QoS and EDF is small.

![Figure 6 Test on overall QoS for different algorithms](image)

5.3 Overall throughput

The present work focuses on QoS performance, but also it is able to provide satisfying throughput performance which is another key performance indicator. Experimental results on overall throughput with different number of LUNs are shown in Fig. 7.

![Figure 7 Overall throughput](image)

As mentioned above, SCAN algorithm has the optimal performance throughput; as an elevator algorithm it only considers movement direction of the magnetic head and the seek address to minimize disk seek time, thus SCAN provides a high I/O throughput. EDF has the worst throughput performance, because it has not considered disk seek address and spends much more time on seeking. iSDS-QoS is able to provide a much higher overall throughput than EDF while ensuring QoS of applications. Besides, iSDS-QoS is close to SCAN.

5.4 Test on real-world I/O workloads

552
In our present work, to test the proposed method, we use two actual I/O traces (block I/O traces) from enterprise servers at Microsoft Research Cambridge [22]. We choose a web server trace which is latency sensitive and a research server trace which has no strict deadline requirements. The two I/O traces are run simultaneously with different LUNs co-located on the shared storage server. Since they are run concurrently, both the I/O workloads have performance degradations due to disk contention, and iSDS-QoS tries to meet the deadline requirements of the latency sensitive application.

The deadline and response time results of web server requests are shown in Fig. 8. It is obvious that, in most cases the response time of the web server workload with iSDS-QoS has lower values than that without the iSDS-QoS - this is because iSDS-QoS dynamically adjusts priorities of the I/O requests. The web server requests were assigned higher priority values to meet their required deadlines. The iSDS-QoS scheduling scheme provided an improvement in the achievement rate of required deadlines increasing it from 36.7% to 83.3%.

5. CONCLUSIONS

To achieve a QoS differentiation in I/O performances in SDS, the present work proposes a scheduling algorithm incorporated with iSCSI storage controller that dynamically predicts the latency for all concurrent I/O applications based on the system status. Based on these predictions the iSDS-QoS assigns and adjusts the priority of the applications. Next the block-level scheduler takes the priority values into consideration while scheduling I/O requests to the underlying disk device. The scheme describes a latency-based QoS-aware I/O scheduling framework above the operating system's disk scheduler for shared storage in SDS environments. It ensures that the latency-sensitive applications won't suffer unpredictable delays in concurrent I/O situations. It also provides a significant improvement in the deadline satisfaction and delivers a QoS guarantee. Experiments on IOMeter and real-world I/O traces verify that the iSDS-QoS can make latency sensitive applications satisfy their required deadlines and guarantee their quality of service.

The present work concentrates on quality of service in IO scheduling, but this is not sufficient for applications under SDS environment. Therefore, in the future we will study further about IO scheduling in SDS, such as how to improve scheduling efficiency with flash memory.

Acknowledgements

Supported by the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDA06010302), and Innovation Institute Foresight Program (Grant No. Y555021601).

REFERENCES


