intelliQoS: Rethinking Storage QoS Implementation for System Efficiency

Yuehai Xu, Marc Patton, Michael Devon Moore, and Song Jiang
The ECE Department, Wayne State University,
Detroit, MI, 48202
Email: {*yhxu, marc.patton, michael.moore9, sjiang*}@wayne.edu

***Abstract*—The objective of maintaining a high efficiency for a shared storage system often has to be compromised with the enforcement of Service-level Agreement (SLA) on quality of service (QoS). From the perspective of I/O scheduling, I/O request service order optimized for disk efficiency can be substantially different from the order required for meeting QoS requirements. When QoS takes priority, the storage system has to serve requests with a sub-optimal efficiency. In this paper, we propose to relate QoS requirements specified for I/O requests to users' experiences. By assuming that only user-observable QoS is necessary for the system to fulfill, we relax QoS requirements on the storage system as long as such a relaxation is not noticeable to users. The relaxation produces a leeway critical for the I/O scheduler to improve disk efficiency. We implement a prototype system, named as *intelliQoS*, as a proof of concept. In the system the scheduler is allowed to schedule requests in its preferred order as long as the user does not sense any performance degradation through the outputs from the application. In this way the user can still experience the same service quality as required although individual requests' latency requirements can be missed for higher storage efficiency. Our experiments on Xen virtual machines (VMs) show that intelliQoS significantly improves system efficiency by up to 80% without violating user-observable QoS requirements.**

***Index Terms*—Service-level Agreement (SLA); Quality of Service (QoS); I/O Request Scheduling; Spatial Locality.**

I. Introduction

Consolidation of I/O services on a shared storage system makes economic sense as it leads to better utilized system resources and reduced management cost. However, a major concern of the shared use of storage system is the reduced system efficiency due to interference among different workloads. Different from the overhead associated with shared use of other system resources such as processor and network, the overhead due to interleaving service of requests between different users' workloads on the hard disks can be significant. As we know, spatial locality is the key to the disk efficiency and the requests from the same application usually exhibit stronger locality than those from different applications. From the perspective of I/O request scheduler, effective exploitation of the locality requires continuously dispatching of requests from the same application for an extended period of time before serving a batch of requests from another application. In this way the long-distance disk seeks can be minimized, and the efficiency of the storage system would not be severely reduced by the sharing. However, this effort on the amelioration of interference for higher disk performance can be foiled by enforcement of QoS policies.

There have been efforts on providing support of QoS differentiation among requests from different applications on production systems. QoS requirements can be specified using either latency or throughput. A deadline, derived from latency or throughput requirement, is usually attached to individual requests for the scheduler to meet in the implementation of QoS. The reluctance of directly using throughput as a metric in the scheduling is due to its lack of clarity on the time period in which a required throughput is measured. For example, a 10MB/s requirement might be interpreted by a scheduler in its implementation as 10MB for every second or as 600MB for every minute. For the latter case, the throughput measured in some 1-second time periods can be much smaller than 10MB/s, which may not be what is expected by the user who sets the requirement. In the meantime, implementation of the throughput in a larger time window, such as one minute, gives a larger space for the scheduler to improve disk performance, or a QoS requirement specified in a larger time scale would dictate less deviation from the request service order optimized for disk efficiency.

Unfortunately today's I/O schedulers are not aware of the time scale against which a throughput can be applied for implementing QoS requirements without compromising user-observable performance. Consequently, either they have to abide by the deadline specified for each request, leaving limited room for optimizing request scheduling for their locality [3], or they would take risk of missing the deadlines by batching multiple requests of stronger locality and higher disk efficiency [2]. In both cases, the space for locality exploitation is limited because the deadlines attached to individual requests are usually too tight to accommodate a sufficiently large batch of requests for being serviced together.

II. Our Solution

Our proposed solution for achieving both QoS and efficiency goals is based on the facts that (1) QoS on the storage is measured according to user-observable service quality, and (2) only I/O QoS that can negatively affect user-observable QoS is significant in its implementation. Accordingly, a request's deadline can be extended as long as it does not cause a user-observable QoS violation, and the scheduler does not have to always immediately service requests of imminent

deadlines. The key to realization of the idea is to determine what requests' deadlines can be extended and how much the extension can be.

A user usually receives its service from a service provider as shown in Figure 1. The user first sends a request to an application. This can be a transaction request for browsing a product catalog or a submission of a job for running at the server. Then the application processes the transaction or executes the job, during which I/O requests are generated and outputs to the user are produced to inform the user of running results or progress. Users perceive service quality only through the outputs, including the timing at which they are produced. The outputs can be used to delimit the requests issued by an application into groups. We can consider any two consecutive outputs to form a container holding a group of requests on which a throughput can be defined and measured. The deadline of the container is defined as the time for its latter output to occur. The premise of our design is that as long as a container's deadline is met, missing deadlines of individual requests in the container does not change user-observable QoS. In this work we use the concept of container to design and implement a system for intelligently implementing QoS, named as *intelliQoS*. In the system, outputs are detected and associated with their requests to form containers, and *intelliQoS* uses the container as scheduling unit when locality can be effectively exploited, and as the unit for the QoS enforcement.

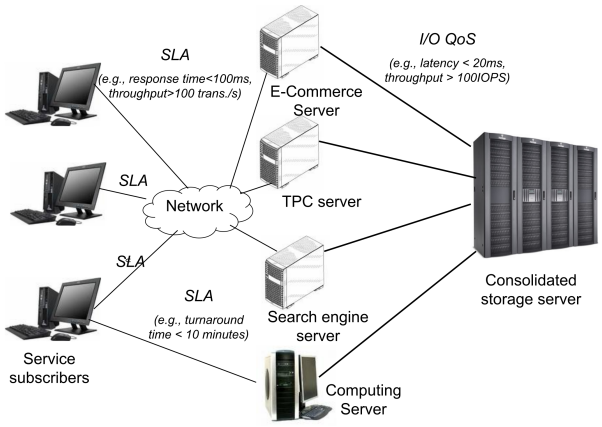


Fig. 1: There are three tiers in a typical system architecture involving consolidated storage service: service subscribers, application servers, and storage server. Service-level agreement (SLA) specifies the QoS requirements on the applications at the application servers, which further determine I/O QoS requirements accordingly to the storage servers. The application servers can be co-located with storage servers. For example, they are virtual machines hosted on the servers attached to a SAN.

III. Design of *intelliQoS*

A generic system architecture that the design of *intelliQoS* assumes is depicted in Figure 1. In the three-tier architecture, the output refers to messages from the application servers to the service subscribers, including system administrators. There are two major components in *intelliQoS*, which are container forming and container-aware scheduling.

A. Forming Containers

Forming container is actually to relate I/O requests to the outputs, or to determine what requests have to be completed for an output to be produced. Usually synchronous requests issued by a process (or thread) are in a container associated with their immediately following output issued by the same process. However, there can be cases where dependency exists between requests and outputs issued by different but synchronized processes. The OS kernel can observe requests and outputs, and can discover their relationship with processes. When application servers and storage servers are not on the same physical machine, the information facilitating identification of containers need to be passed from the application servers to the storage servers. To this end, we piggyback a unique process identification to each request sent to the storage server, and create a fake request for each output with its associated process identification to notify the storage server of the occurrence of the output. The overhead of communicating the additional information is small.

Another typical system configuration is that application servers are virtual machines co-located with the storage server on the same host, which is connected with storage devices. The hypervisor or host VM in a VM system can monitor any requests to the storage and any output messages to the clients through NIC, and can associate outputs to requests at the granularity of guest VMs without any instrumentation of guest OS. That is, requests from the same guest VM are placed into containers according to occurrences of this VM's outputs. This is certainly a conservative option as one process's container may have to be made smaller due to outputs from other unrelated processes on the same VM. A smaller container limits the potential of performance improvement of the proposed approach. To overcome this, guest OS has to be instrumented to discover the association between requests (or outputs) and processes on the same guest OS. As a quick proof-of-concept prototype of using container for efficient QoS implementation, *intelliQoS* is built for the virtual machine environment with co-located application servers and storage systems and with containers formed at the level of VMs.

B. Container-aware Scheduling

If the size of a container is substantially large and its requests have strong locality, it is expected to significantly reduce disk head thrashing between requests from different applications by scheduling requests in one container at a time. We assume that each request has been assigned a deadline by another system component to ensure that user-observable QoS is satisfied if the deadlines are met. As there are no outputs within a container, the deadlines of all requests in the container can theoretically be extended to the container's last request's deadline without compromising user-observable

QoS. In practice, the extended deadlines can be spaced with a short time gap to account for request service times. To this end, we need to estimate container size, or the number of requests in a container, and the deadline of the last request in the container. For the prediction, we assume that there is a stability on container size so that we can use history container sizes to statistically predict next container's size. The formula for the prediction of current container size ($container\_size_k$) is borrowed from the Linux kernel where it is used for prediction of thinktime in the anticipatory scheduler:

$sample_0 = 0;$
$total_0 = 0;$
$sample_k = (7 * sample_{k-1} + 256)/8;$
$total_k = (7 * total_{k-1} + 256 * container\_size_{k-1})/8;$
$container\_size_k = (7 * total_k + 128)/sample_k;$

In the formula both recent and history statistics are considered to smooth out short-term dynamics, and to phase out historical statistics by giving recent statistics a higher weight. It is noted that when $container\_size_k$ is to be predicted $container\_size_{k-1}$ is a measured value. For each container we also track its average allowed-latency, which is the time period from the beginning of the container to the deadline of its last request divided by the container size. We use a formula similar to the one used for predicting container size to estimate the average allowed-latency for the current container. Therefore, the deadline of the last request for deadline extension can be estimated as the product of the average allowed-latency and the container size.

When the next output occurs, it may turn out that the estimated container size can be larger or smaller than the actual container size. In the former case, user-observable QoS can be compromised. To minimize the risk, *intelliQoS* sets up a safety zone, which is one third of estimated container size by default. That is, the container size used for deadline extension is only 2/3 of the estimated container size. In the latter case, the full potential of container-aware scheduling is not exploited. Note that *intelliQoS* only extends the deadlines of requests before the assumed last request in a container. For the ones between the last request and the actual last request, we conservatively keep their deadlines unchanged. As a future work, we plan to study the tradeoff of risks and benefits of aggressively extending request deadlines for different applications in terms of user performance experience and system efficiency.

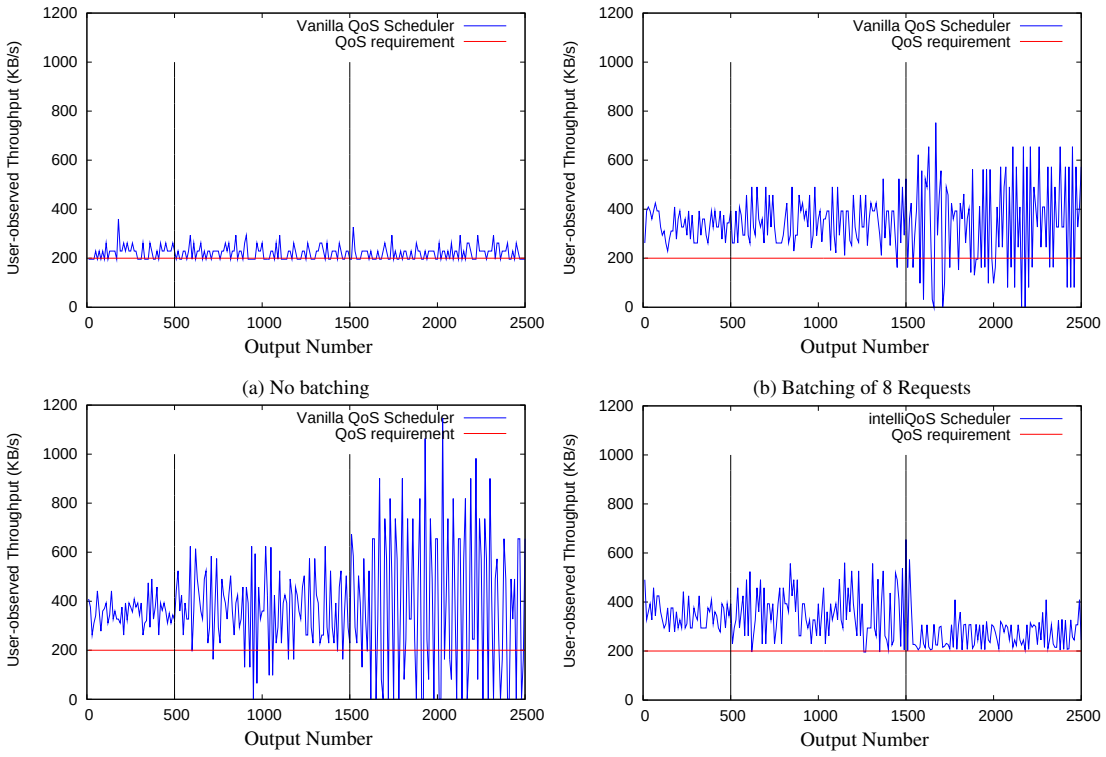
IV. Performance Evaluation

The *intelliQoS* prototype is implemented on the Xen virtual machine (Xen-4.0-testing) with Linux kernel 2.6.32.23. In the host domain we implement a request-deadline-aware I/O scheduler, much like the CVC scheduler proposed in Stonehedge [3]. The scheduler is adapted from the Linux deadline scheduler. It dispatches requests according to their CSCAN-determined order unless requests with imminent deadlines emerge and it switches to the deadline-driven scheduling. This scheduler is the vanilla one against it *intelliQoS* is compared. If *intelliQoS* selectively extends request deadlines, the scheduler works as intelliQoS's scheduling component. VMs are hosted on the server of two quadcore processors (Intel Xeron5450) and 8G RAM. The data of all VMs are stored on a Seagate SATA hard drive (ST3500514NS) of 7200 RPM and 500GB capacity with a 32MB built-in cache. Each VM is configured with 1G memory and two vCPUs. The I/O scheduler for the guest VMs is NOOP. Blktap2 is used to pass the I/O requests from guest VMs to host VM, which does the actual dispatching of I/O requests to the disk.

A. Adapting Storage Efficiency to User-observable QoS

In this experiment, we investigate how user-observable QoS can be affected by attempts for improving storage efficiency. The benchmark used in this experiment is FIO-1.5.0, a popular and highly configurable I/O benchmark. We configure it to asynchronously read 4K random data from a 128MB file with its *ioengine* parameter set as *libaio*. The *iodepth* parameter is set to 100, which means that we allow at most 100 requests in flight. In the system there are two VMs, each runs on an FIO benchmark. The I/O latency requirement on each request is set to 20ms for both VMs, or 50 IOPS for each VM, which is equivalent to a throughput of 200KB/s. The files read by the two FIOs are separated by a space gap of 32GB on the disk. To simulate different intensity of interaction between the application and the users, we change frequency of calling *printf()*, which is used in FIO to report throughput measured between two consecutive outputs to users, during each run of the benchmark. Figure 2 shows user-observed throughputs (reported by FIO's *printf()*) with different QoS-aware schedulers. While batching can be used for improving disk efficiency, we show the throughput with the vanilla QoS scheduler that allows requests from the same VM to serve in a batch of 8 (Figure 2-b) or a batch of 16 (Figure 2-c).

As shown in the figures, the user-required QoS can be satisfied by using the vanilla scheduler due to its effort on meeting individual requests' deadlines. However, its throughput stays only a little above required 200KB/s throughput. When we increases the batching size to 8 and then to 16, the average throughput improves by almost up to 100%. However, because the batching blindly groups requests for scheduling without regard of its implication on user-observable throughput, the user-observable QoS requirement (200KB/s) is more and more frequently and severely violated with the increase of batching size and output frequency. In contrast, intelliQoS tracks container sizes determined by output occurrences and intelligently groups requests for scheduling according to the sizes. The adaptivity built in the QoS-aware scheduler allows the scheduling to opportunistically take advantage of "blind spot" in users' performance experience for higher storage efficiency without QoS violations. Note that while the throughput improvement beyond what users ask for may not seem interesting, the corresponding efficiency improvement would be valuable for other purposes such as accommodating more workloads in a production system.



(a) No batching

(b) Batching of 8 Requests

(c) Batching of 16 Requests

(d) intelliQoS

Fig. 2: User-observed throughputs for the first 2,500 outputs of one FIO instance using different QoS-aware schedulers. (a) The vanilla QoS-aware scheduler meeting each request's deadline; (b) The vanilla scheduler with batching of 8 requests; (c) The vanilla scheduler with batching of 16 requests; and (d) intelliQoS. The throughputs reported by the outputs are shown in three stages delimited by vertical lines, as output frequency is changed at those moments. The frequencies are 1, 2, and 5 outputs/s, in this order.

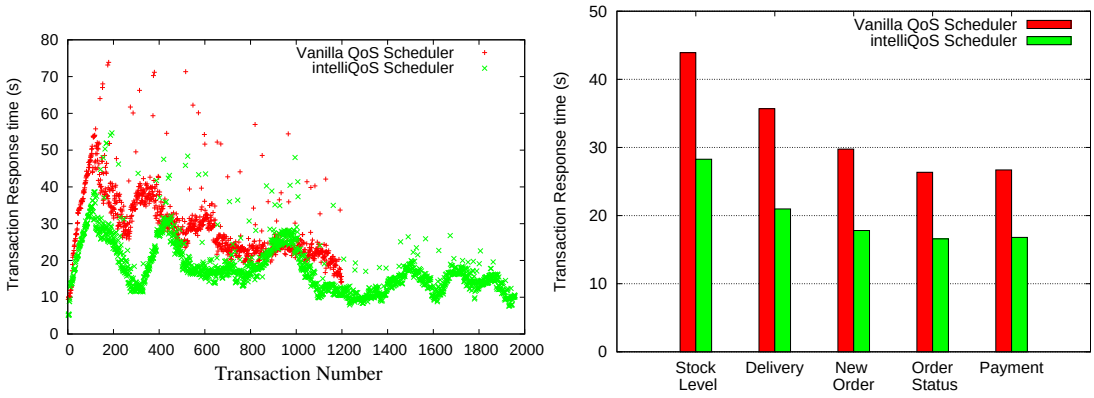
B. Experiments with Macrobenchmark

TPC-C is a standard benchmark to measure the performance of a transaction processing system. We use MySQL 5.0.22 as the database server and dbt2-0.40 to create tables in it. We choose the number of warehouse as 10 to generate the database. Same with the experiment with FIO, we set up two VMs of the same QoS requirement (50ms latency for each I/O request). Each VM runs a TPC-C server, which is connected to a client via a gigabytes network. Each client has 16 connections to its TPC-C server and the test time is set as 5 minutes. We use TShark 1.0.15 to capture and analyze network packages to discover outputs to each client. The user-observable performance, or the transaction response time, is shown for the entire run of one TPC-C instance (Figure 3-a) and for each category of transactions (Figure 3-b). As shown in the figures, intelliQoS can significantly reduce the response time for transactions in different categories. For example, for new-order transactions, the time is reduced by 42%, or its NOTPM (number of transactions per minute) is increased by 39% (from 110.7 to 181.0). In Figure 3-a, for transactions of same type, which can be recognized by comparing shapes of the two curves, intelliQoS always produces smaller response times. Assuming the vanilla scheduler can meet the user's QoS requirement, intelliQoS meets the requirement with a much higher efficiency.

V. Related Work

The commonly used metrics for specifying QoS requirements to storage service are throughput and latency. A deadline associated with each request is computed from the specification. It is then attached to the request as a tag for the I/O scheduler to decide a service order accordingly [1]–[4]. Sometimes the deadlines can be adjusted according to the load of the system. For example, in the pClock scheduling, the deadlines can be adjusted by the request burstiness (the number of pending requests) and request arrival rate [1]. In these works the setting of requests considers conformity with the QoS specified in the SLA, but does not consider giving more space for the I/O scheduler to exploit locality as intelliQoS does. As soon as intelliQoS relates request deadline to the user-observable QoS, the extended deadlines readily facilitate improving disk efficiency by using schedulers considering both QoS and locality such as CVC in [3].

A work that is close to intelliQoS in spirit is external synchrony [5], which asynchronously implements synchronous



(a) (b)

Fig. 3: (a) Transaction response times with the vanilla QoS scheduler and intelliQoS during the execution of one TPC-C instance. Because the two runs are of a fixed runtime, intelliQoS completes more transactions. (b) Average response times for transactions in different categories.

I/O for high disk efficiency and keeps synchrony for external observers. This is achieved by correlating the write-back operations with the outputs to users. IntelliQoS takes a similar approach but for a different purpose – retaining locality while providing *external* QoS guarantee. Furthermore, intelliQoS faces a bigger challenge – it has to predict when the next output would occur.

VI. Conclusion and Future Work

In the paper we propose the concept of container that captures both user-observable QoS and locality, and investigate its feasibility on achieving both external QoS guarantee to users and high I/O efficiency with a prototype implementation in a VM system. The preliminary results are promising and encouraging. Our future work plan includes investigation of how accurately the container size can be predicted in various application scenarios, the implication of aggressively extending request deadlines, and implementation in a distributed environment.

Acknowledgment

This work was supported by US National Science Foundation under CAREER CCF 0845711 and CNS 1217948. We thank the anonymous reviewers, who helped improve the quality of the paper substantially.

References

[1] A. Gulati, A. Merchant, and P. J. Varman, "pClock: an arrival curve based approach for QoS guarantees in shared storage systems," *in Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2007.

[2] A. Gulati, A. Merchant, P. Varman, "mClock: Handling Throughput Variability for Hypervisor IO Scheduling," *in Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[3] L. Huang, G. Peng, and T. Chiueh, "Mutli-dimensional storage virtualization," *in Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2004.

[4] C. Lumb, A. Merchant, and G. Alvarezg, "Facade: virtual storage devices with performance guarantees," *in Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[5] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn, "Rethink the Sync," *in Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.