Performance virtualization for large-scale storage systems

David D. Chambliss Guillermo A. Alvarez Prashant Pandey Divyesh Jadav
Jian Xu* Ram Menon* Tzongyu P. Lee†
{chamb,alvarezg,ppandey,divyesh,jxu,menon}@us.ibm.com,tpl@ieee.org
IBM Almaden Research Center, 650 Harry Rd., San José, California 95120, USA

Abstract

Current data centers require storage capacities of hundreds of terabytes to petabytes. Time-critical applications such as on-line transaction processing depend on getting adequate performance from the storage subsystem; otherwise, they fail. It is difficult to provide predictable quality of service at this level of complexity, because I/O *workloads are extremely variable and device behavior is poorly understood. Ensuring that unrelated but competing workloads do not affect each other's performance is still more difficult, and equally necessary. We present* SLEDS*, a distributed controller that provides statistical performance guarantees on a storage system built from commodity components.* SLEDS *can adaptively handle unpredictable workload variations so that each client continues to get the performance it needs even in the presence of misbehaving, competing peers. After evaluating* SLEDS *on a heterogeneous mid-range storage system, we found that it is vastly superior to the raw system in its ability to provide performance guarantees, while only introducing a negligible overhead.*

1. Introduction

Data formerly stored in a myriad of direct-attached storage (DAS) units are currently being *consolidated* into large data centers, to reap the benefits of statistical sharing and lower management costs. Resulting data centers may contain hundreds of hosts requiring a terabyte or more of data each, stored in large, shared storage servers interconnected by a storage area network (SAN) such as Fibre Channel (FC).

Consolidation introduces additional coupling when previously unrelated workloads begin to share resources such as disk drive actuators, network links and endpoints, switch backplanes, controller processors, data caches, system buses, and SCSI interconnects. Resource pooling weakens the assurance of acceptable performance to any given application, because of competition with others outside of its control.

Applications with absolute performance requirements will fail if the storage subsystem does not provide the required quality of service (*QoS*): penalties or losses may be incurred if financial transactions are not processed before contractual or statutory deadlines, and customers may be lost if a web ordering system's response is too slow. Approaches based on overprovisioning or static resource partitioning have proven too inflexible; application-level adaptation to the available QoS is limited to code written with that purpose in mind, and thus restricted to a few niche genres.

This paper describes the Service Level Enforcement Discipline for Storage (SLEDS) system, a production-capable outboard storage controller that manages client workloads to meet QoS goals. SLEDS provides virtual slices of system performance, as governed by statistical QoS guarantees, in such a way that the performance experienced by each client is not affected by other clients' behavior. Guarantees of throughput and response time are expressed in absolute terms. The result is *performance virtualization*: the client experiences storage performance at least as good as if a subset of physical resources were dedicated to it.

SLEDS's goal is to ensure that all guarantees are met, as long as enough aggregate resources exist. It accomplishes that by taking periodic performance samples, and by *throttling* (delaying) I/Os from overly-demanding clients whenever other clients experience inadequate performance. SLEDS is transparent to the clients and back-end devices (e.g., RAID storage servers, disk enclosure controllers, or even individual hard disks) with which it interacts, for it uses a separate gateway device to intercept I/Os. The system is rapidly responsive to changes such as sudden workload fluctuations and device failures. The SLEDS system presented here does not attempt to solve the problem of admission control (i.e., determining whether the storage system can accommodate the performance demands of a new client or application, while meeting existing commitments), nor that of provisioning (identifying and allocating an appropriate resource to meet new or changed commitments).

We describe some related work in Section 2, and present a black-box description of SLEDS in Section 3. Section 4 describes how individual components in our architecture operate, with some comments about the current implementation in Section 5. We evaluate the performance of the SLEDS prototype in Section 6, and present some conclusions in Section 7.

*IBM Systems Group, 5600 Cottle Rd., San José, California 95193, USA.
†Now a private consultant.

2. Related work

Performance guarantees based on physical separation (e.g., direct-attached storage) and overprovisioning can be a particularly expensive proposition for storage: worst-case resource consumption is orders of magnitude higher than the average for bursty workloads. Static resource partitioning (e.g., zoning in FC SANs [8]) is only marginally more flexible, and results in low system utilization. Existing tools for one-stop performance monitoring and trend analysis [6, 13] leave humans to decide when and how to act when problems occur. SLEDS goes a step further by taking immediate, automatic corrective actions.

An alternative approach is to have upfront knowledge about expected traffic loads, and use it to design a storage system capable of providing the required performance [1, 2]. The quality of the resulting design is highly dependent on the accuracy of this knowledge—which is not available in many practical situations. The process can be repeated to refine the data placement later in the lifetime of the system, but it cannot approach SLEDS's very fast corrective actions; data migration typically takes hours to days to complete.

Many networking solutions [4] are based on selectively dropping packets [5]. They do not extend to widespread storage access protocols [3] for multiple reasons [18], including the severe consequences of packet loss. Proposals like DiffServ [5] are not rich enough to distinguish all service classes that may need to be treated differently.

Only a few niche applications can adapt to variable storage QoS. Existing solutions for multimedia servers [11] are also based on self-scheduling, but assume predictable workloads (e.g., read-only, homogeneous streams) and large client buffers. Unlike Cello [16], SLEDS does not depend on predictive storage device models, treats separate QoS levels independently, and handles load variations in a fully adaptive way. YFQ's [7] centralized scheduler provides relative guarantees on a device's resources. Guaranteed fractions of an unknown, hard-to-characterize totality of resources are not realistic or useful; SLEDS provides absolute guarantees with a more scalable, distributed architecture.

Façade [14] provides per-workload storage performance monitoring and QoS enforcement capabilities. A central scheduler regulates the rates of I/O workloads accessing a common storage container such as a RAID logical disk. Façade does not account for competing workloads sharing resources in various degrees, e.g., two logical units (LUs) in the same vs. in different disk arrays; nor can it associate incoming I/Os to their service classes in the general case. In contrast, SLEDS can manage very large systems by virtue of its decentralized QoS enforcement architecture, and can classify and process I/Os with very low overheads because of its careful design and implementation. Most important, SLEDS (unlike Façade) can throttle just the workloads that are responsible for performance degradation—as opposed to indiscriminately decreasing the QoS experienced by every workload in the system.

3. Objective

Most current storage systems are designed for best-effort service. Barring crude overprovisioning, it is not possible to provide deterministic QoS guarantees in the presence of unforeseen events such as workload variations and failures. SLEDS provides *statistical* guarantees to clients by dynamically reacting to such events.

SLEDS's goal is to improve the *effectiveness* of the storage system, as opposed to improving standard aggregate performance metrics. This is done by adaptively redirecting resource away from applications that are receiving adequate QoS or better, toward those experiencing bad performance. For example, throttling may well increase average system-wide response times, if that is deemed necessary to satisfy more clients. In particular, SLEDS performs resource arbitration in a SAN-based system for block-level storage service, using SCSI commands to access data.

A *stream* is the basic unit on which performance is measured and controlled; every I/O in the system belongs to some stream. A SLEDS stream consists of all I/Os originated at the same SCSI [3] initiator (i.e., the same host bus adapter) and targeted at the same device and LU. SCSI devices may be physical or logical (e.g., disk array LUs, or data containers exported by virtualization appliances [9, 10, 12]).

Service Level Objectives (SLOs) describe the performance requirements of streams. A SLEDS SLO specifies bounds on delivery and demand. Delivery targets bound the average response times the storage system must deliver during a time window, as long as the demand metrics (such as IOPS or bytes/s) are not exceeded. Exceeding the demand limits may result in arbitrarily bad QoS without violating the SLO. The delivery guarantees may be missed in at most a specified, small, percentage of windows. SLOs may also specify penalties for breaking the agreement. Each stream is associated to a single SLO, although a group of streams (a *service class*) may share an SLO. Bounds on demand apply to the whole group, without restrictions on the relative allocation to each member; bounds on QoS similarly apply to average group performance.

SLEDS's main inputs (Figure 1) are the QoS specifications created by human administrators, or gathered from a library of common workloads, or automatically generated by an online monitoring tool. SLEDS may optionally accept a description of the system's configuration, including the interconnection pattern between components. This helps quantify the extent to which streams compete for resources, to make more accurate automatic throttling decisions; throttling a stream that shares no resources with an underperforming client has no beneficial effect.

SLEDS intercepts all I/O commands for monitoring and performance control. System administrators can access monitored performance metrics in the form of reports (e.g., for trend analysis), and issue manual commands that override SLEDS's automatic resource arbitration decisions.

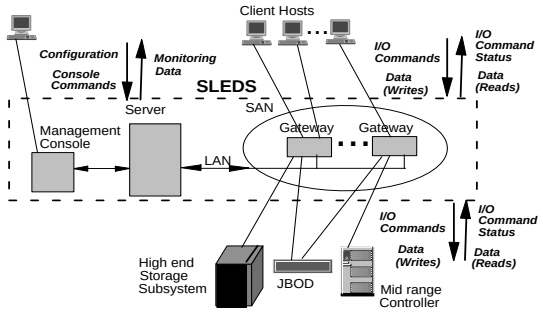


Figure 1. SLEDS **Architecture.**

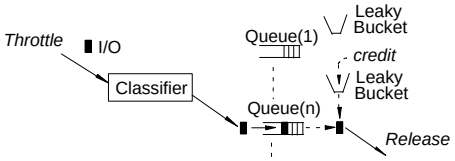


Figure 2. Use of leaky bucket mechanism for throttling. The replenishment of credits for a given leaky bucket at a set rate is triggered by time and I/O activity.

4. Functional description

Figure 1 shows SLEDS's components and their main interactions. One or more *gateways* receive I/Os (SCSI commands) from client hosts and relay them to the back-end storage devices. The gateways gather performance statistics and selectively throttle streams. The SLEDS *server* tracks the performance being experienced by each service class in the system (as reported by the gateways) against its QoS requirements. If one or more classes are not receiving the QoS they need, the server instructs the gateways to throttle other streams. The server interacts with a potentially-remote *management console*, where system administrators can peruse the historical and current system performance and issue manual throttling commands. The server communicates over IP; in particular, it issues remote procedure calls (*RPC*s) to the gateways.

Performance and scalability are primary objectives, so SLEDS architecture separates the scheduling actions in the gateways from strategy decisions in the server. This may be abbreviated as "think globally, act locally." I/Os entering a storage system may be distributed over many nodes, and streams entering via different nodes may compete for resources and interfere with one another's QoS. A good throttling strategy therefore requires global knowledge. However, inter-node serialization or synchronized update of global state on every I/O would make it nearly impossible to achieve required processing rates well above 10,000 I/Os per second per processor.

In SLEDS, the queue and release decisions for an I/O depend only on the parameters of its stream; they are made without any current knowledge of requests from other streams. This is compatible with the design of state-of-the-art storage subsystems, in which the processing of individual I/Os is decoupled as much as possible. The SLEDS server is the central analysis component that takes into account the coupling among streams and orchestrates all enforcement points. Scaling up to many streams and clients is not limited by I/O-path considerations.

4.1. Gateway

This and following subsections describe in more detail the operation of each SLEDS component. In our design, a gateway consists of two main components. The *IO logic* is the base function for receiving and forwarding concurrent I/Os to the back-end, relaying the data and status as necessary. In a production system, the IO logic would incorporate functions such as LU virtualization, online data migration, and remote copy. (Alternatively, the IO logic could be the data path of a SAN switch in order to incorporate gateway function into the network fabric.) The IO logic interacts with the *IO intercept*, the code added by SLEDS for monitoring and control. The IO intercept decides which individual I/Os are to be throttled based on local information, updated periodically by the server.

The IO intercept's operation is shown schematically in Figure 2. It classifies I/Os by stream, and regulates the rate at which each stream's I/Os are serviced using a version of the leaky-bucket admission control policy [17]. Each stream owns a queue and a set of buckets. Each bucket holds tokens representing rights to occupy a resource. There are different buckets for protocol processing, disk seeks, and data transmission resources, and for reads and writes. When a stream is throttled, its I/Os pass through its queue before being released to the back end, in order, as tokens become available. When the buckets contain enough tokens for the resource cost of the head I/O, that I/O is released and tokens are removed from the buckets accordingly. Tokens are supplied into buckets continually according to that stream's usage limits, so every I/O is eventually released. Each bucket has limited capacity so that surpluses do not build up in inactive periods. This mechanism regulates the rate at which each stream consumes resources, but it must be carefully implemented. Our implementation reduces overhead by performing the supply calculations only as needed to release I/Os at correct times.

Once an I/O has been released from the throttling delay queue, it is serviced as it would be without workload controls, except for timestamping to track response times. The use of the system is transparent to client hosts and applications, except that the address of data on the SAN is mapped to one on the gateway. A throttled I/O stream is perceived

just as slower performance for that client's slice of the storage system.

4.2. Server

The server keeps the global state and configuration information for the whole storage system. It receives the SLOs for each service class from the management console, and may also get information about the system's topology and device capabilities. It makes the high-level decisions on resource allocation, to be carried out by the individual gateways. Its position as an omniscient observer allows it to take all factors into account in order to satisfy SLEDS's goals. System administrators (or automatic management tools) can submit commands such as policy changes, that have direct impact on the high-level decision making.

The server periodically queries gateways for performance data. Individual and aggregate measurements are stored in a database for online and historical analysis. The server compares recent performance to the SLO for each service class. It must throttle more streams when either (a) one or more service classes are determined to be receiving inadequate QoS (i.e., excessive I/O service times), or (b) the console issues a manual throttling command. Similarly, throttle limits may be relaxed or removed as a result of automatic decisions made by the server as well as manual commands.

It is very difficult to compute directly the best throttle limits for a given situation. That would be NP-hard even with perfect knowledge about future workload variations and with accurate device modeling—and neither is available. The SLEDS server instead applies an incremental heuristic, relaxing limits on streams whose SLOs are not met, and tightening by steps the competing streams which can best afford to have throughput reduced. The first streams throttled are those whose throughput exceeds the limits in their SLOs, particularly those whose latency objectives are met nonetheless. Streams with moderate demand may also be throttled as long as their SLOs are still met. When no QoS violations are happening, throttle limits are relaxed gradually until they must be reduced again to protect QoS on other streams. Other heuristics are under study, but this one has been adequate in our experiments.

4.3. Management console

The user interface allows a customer to create service classes and their SLOs by completing forms. The administrator can specify storage resources by defining individual streams, and optionally clustering them into disjoint groups. Separately, he may create SLOs that are later bound to their respective service classes. The management console also allows administrators to enable or disable automatic throttling, and to enter manual throttling commands. It provides a comprehensive set of live reports on the system's performance, that can be used by humans to base their decisions.

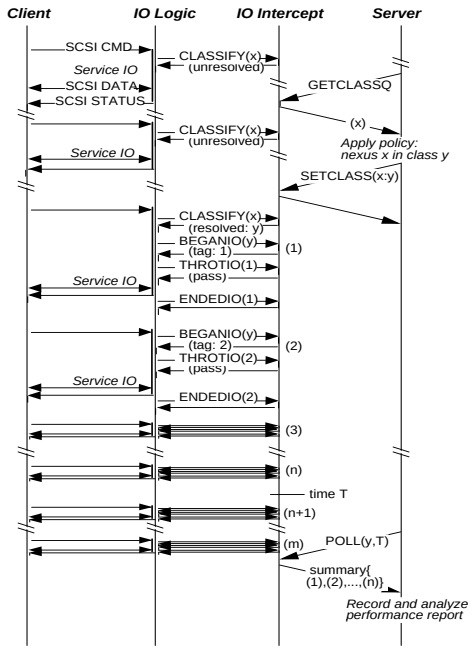


Figure 3. Operation of classification and monitoring. The IO logic and the IO intercept execute in the same gateway node. The lifetime span of each outstanding I/O in SLEDS is shown by a vertical line to the left of the IO logic timeline.

4.4. Steps of I/O processing

The operation of the system for monitoring and throttling is illustrated in Figures 3 and 4. The following description applies to throttling commands regardless of whether they were triggered by our automatic heuristic or by the management console.

Before an I/O is subjected to flow control, it must be classified as belonging to a service class. IO logic uses a CLASSIFY call to request the control class for a SCSI I_T_L *nexus* [3] (here designated as x). The server periodically retrieves pending classification requests (GETCLASSQ RPC), performs classification by the rules it has received, and installs service class parameters (here designated y) and the classification mapping (x:y) (SETCLASS RPC). During this asynchronous process, any I/Os to the as-yet-unclassified nexus are associated with a default service class, and processed. I/O processing is never blocked pending communications with the server. Once a classification is resolved, the IO logic caches the result and applies it to future I/Os.

Each classified I/O in the IO logic is registered with the IO intercept with the BEGANIO call, which returns a token so subsequent calls can associate event timestamps with the same I/O. The calls include ENDEDIO when the I/O has completed and delivered status, and THROTIO which determines whether the I/O should be processed immediately or

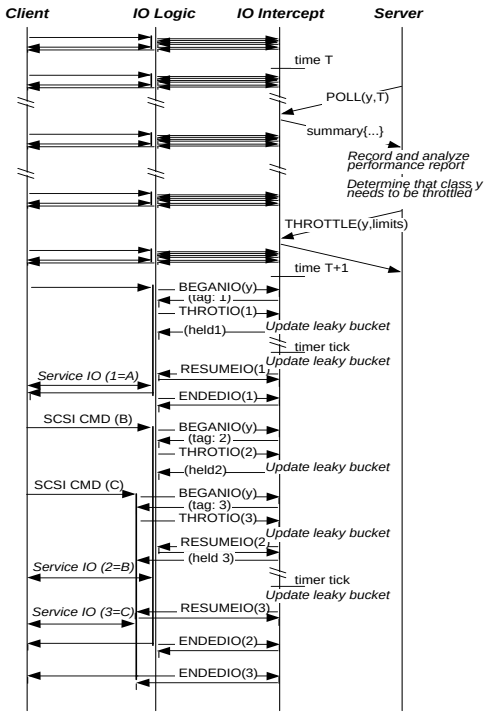


Figure 4. Operation of throttling. Upper-case letters A, B, C **identify** SCSI I/O**s, numerals** 1, 2, 3 **the tokens** SLEDS **assigns to them, and** x **and** y **their nexus and service class.**

delayed. No throttling is in effect in Figure 3, so THROTIO merely records the I/O's attributes and allows it to proceed. The calls are labeled explicitly only for the I/Os assigned token values 1 and 2, but they occur also for the I/Os shown with unlabeled arrows.

The IO intercept's timestamp data are aggregated into summaries which are periodically retrieved by the server (POLL RPC) and saved and analyzed. While the server works with data from a past interval, the IO intercept records and summarizes data for subsequent intervals. For simplicity, Figure 3 illustrates only one nexus and class and only one request at a time. In a real system hundreds or thousands of nexuses, classes and concurrent requests may be handled together.

Figure 4 shows the application of throttling. The server's decision to throttle may be driven by manual action, by standing policies, or by automatic adjustments. In the illustrated scenario, the THROTTLE RPC installs limits for class y after monitoring results ending at time T have been analyzed. Throttling is latched in at time T+1. After this, each call to THROTIO invokes leaky-bucket processing. The I/O is queued as in Figure 2, and the leaky-bucket data structures are updated. Leaky-bucket updates can also be triggered by timeouts. However triggered, an update adds tokens to buckets in proportion to the time elapsed since the last update. I/Os are then released from the head of queue as long as enough tokens are available. When an I/O is released, the IO logic is notified via the RESUMEIO callback function. In Figure 4, the sample timing is such that the requests with tokens a and c are released on timer-driven updates, whereas b is released by the update triggered by the arrival of c.

4.5. Reliability

The SLEDS goal of maintaining QoS levels cannot override the mission-critical system goals of I/O correctness and forward progress. To protect the reliability and performance of I/O service, the I/O path in the gateway is decoupled from communications with the server. Processing of an I/O never blocks on any SLEDS communication outside the gateway. The gateway holds no server-connection state aside from transport identifiers and authentication data. In gateway-server transactions, the gateway just processes RPCs that report or change its state. In classification processing, for example, GETCLASSQ is idempotent and does not create a requirement that a SETCLASS will follow. If the stream of RPCs stops for any reason other than a failure of the gateway, the gateway will continue to service I/Os and apply the limits that were in effect. The server is responsible for maintaining any server-gateway joint state. As a result, the gateway consequences of even complex sequences of server failures, restarts, and failovers can be understood and tested in terms of a relatively small gateway state space.

5. Prototype

The SLEDS prototype handles SCSI commands from client hosts on a FC SAN, directed to data on off-the-shelf back-end storage. No added instrumentation or other modification is required either on hosts or on storage devices; SLEDS's presence in the system is transparent. We successfully tested this prototype with heterogeneous clients, running a variety of operating systems, and with a heterogeneous back-end—two different midrange disk arrays. SLEDS can manage the resources of an adequately-provisioned system by knowing only the SLOs, and nothing about the individual capabilities of each component.

We implemented the gateways by adding about 25K LOC (lines of code) of C to a preexisting *IO engine*, a FC storage controller base developed in-house, running on Linux on dedicated x86 machines. For our purposes the IO engine is just an interception point that receives and forwards client I/Os. As noted above, interception elsewhere in the I/O path would also be viable.

The SLEDS IO intercept does not reorder I/Os within a stream. In the prototype, each stream is a single SCSI I_T_L nexus [3] which identifies the host port and gateway port handling the I/Os and the LU to which they are addressed. In the leaky-bucket throttling, the replenishment time that

specifies the fastest accumulation of new credit is typically 2 ms and timeouts every 20 ms is used to ensure that held I/Os do not languish in the queue.

An IP network (separate from the FC SAN for security) carries RPCs between gateways and the server. The SLEDS server contains about 88K LOC and relies upon IBM DB2, a commercial database manager, to store and retrieve performance data.

All SLOs in the system are treated as having equal priority, i.e., equal cost of failing to provide the contracted QoS. Performance data for each stream are reported and used for closed-loop actions at 1-second intervals, and 10-second intervals are used for recording and display on the management console.

6. Experimental evaluation

6.1. Experimental Setup

The experimental setup used for evaluating the performance and effectiveness of SLEDS had three load-generating machines (clients), a gateway, a server and two back-end storage subsystems. The FC SAN fabric was built using an IBM 16-port 2Gb/sec switch and 2Gb/sec host bus adapters on all nodes. The clients and gateway were IBM xSeries 345 servers, with a 2.4 GHz Intel Xeon processor and 512 MB (client) / 2 GB (gateway) RAM, running Microsoft Windows 2000 Server (client) / Linux 2.4.7-10 (gateway). Client load was generated using IOMeter [15]. An IBM xSeries 330 server with a single Intel Pentium III 1.133 GHz CPU and 256 MB RAM running Linux (2.4.18-3) served as the SLEDS server. The back-end storage consisted of two mid-range FC subsystems. Each had 36.4 GB 10K rpm FC (Seagate ST336605FC) disk drives. One subsystem was an IBM TotalStorage FAStT 700 with 40 disks configured to have three LUs : two 6+6 RAID 1/0 LUs and a 16-drive RAID 0 LUs . The other subsystem was an IBM ProFibre 4000R 5840 with 58 disks which were configured into four RAID 0 LUs with 12 to 17 disks.

6.2. Overhead

The added processing required for workload management has the potential to consume enough resources to outweigh the benefits of regulating competition. Thus, an essential part of SLEDS evaluation is to ensure it has minimal impact on workloads that are not intentionally delayed.

The addition of SLEDS capability to an IO engine consists of adding the IO intercept component and calls to it. A first evaluation was to measure elapsed time of all IO intercept calls with an on-processor clock, within an IO logic skeleton that performed no actual I/O, but simulated many streams with I/Os going on in parallel. The result on the gateway machine was a total of 1.69 μs elapsed time per simulated I/O.

Mode	Rand W IOPS	Seq W IOPS	Rand R IOPS	Seq R IOPS
Client queue depth = 32				
Base	10855	11694	9973	24306
SLEDS-mon	10874	11478	9337	22229
SLEDS-throt	10863	11421	9310	21920
Client queue depth = 128				
Base	11076	11652	12803	24450
SLEDS-mon	11097	11669	12518	23979
SLEDS-throt	11053	11665	12499	24205

Table 1. Overhead microbenchmarks of SLEDS. Throughput for random and sequential writes, and random and sequential reads for two different client queue depths (number of outstanding I/Os) is presented. (Values shown are means of results from three runs; the confidence interval for all the means was less than 2%). The Base configuration used the I/O engine, with no modifications for SLEDS; the other two cases used the same SLEDS executable in different modes. In the monitor mode (SLEDS-mon), no streams are subject to throttling; all calls to the IO intercept are made, and monitoring results are communicated to the SLEDS server, but leaky-bucket accounting is not performed. In the throttle mode (SLEDS-throt), throttling was enabled for all streams, with limits in excess of demand, which resulted in full leaky-bucket accounting being done for every I/O.

For operations as short and frequent as calls to the IO intercept, an important factor in processing time is the impact on L1 cache efficiency which can be very different in a real system with actual I/Os and a different number of streams. The impact of SLEDS in a more real environment was evaluated with microbenchmarks, with results in Table 1. Synthetic closed-loop loads from all three clients were applied to all back-end logical disks, so that 98 disk drives were used. Each run applied pure single-block I/Os of a single type (sequential read, sequential write, random read, or random write).

For random reads and writes and sequential writes, the variation among different modes is not very significant. It appears that these rates are limited by the back-end controllers or disks. In any case we can say conservatively that, with throughputs in several modes in excess of 10,000 IOPS, operating SLEDS does not decrease throughput more than 2.5%. For sequential reads, the rate may be limited by the processor running SLEDS (column labeled "Seq R IOPS" in Table 1). Assuming a 100% CPU utilization, the reciprocal of rate is the average per-I/O processing time. When the assumption is false, this serves as an upper-bound on the processing time. Using this method, the base configuration

has a processing time ceiling of 40.9 μs for sequential reads and the corresponding ceilings are 41.7 μs and 41.3 μs for the two modes with SLEDS. So, we can estimate that the increase in processing time due to SLEDS is 0.8 μs, or about 2%. We conclude that adding SLEDS controls has a negligible impact on a storage server's core function.

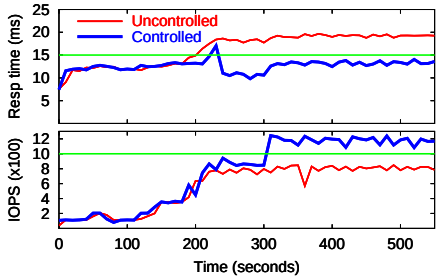
6.3. Performance isolation

The key capability of SLEDS is to regulate resource load so that SLOs are achieved. This has been tested using numerous combinations of request streams issued from different application hosts. An example of the effectiveness is seen in Figure 5. In the uncontrolled case, where no throttling is applied, both workloads meet objectives at first, but when the transaction (TP) demand increases, the backup (BU) load prevents TP from receiving adequate response time, and the system goes into a failed QoS state. The automated imposition of throttling when needed (controlled case) solves the problem. In this scenario, the BU stream is permitted to exceed its allocation, and thus make more progress in its task, until around time $t = 230$s, when the TP demand is high enough that the BU overload is a problem. At that point the BU stream is throttled to the point where both streams are succeeding, as measured by the SLOs.

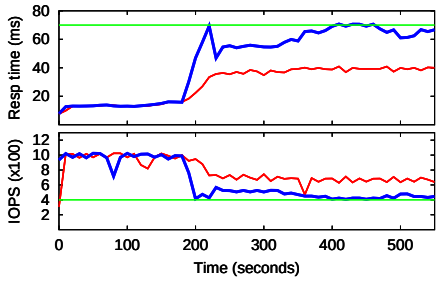
The synthetic workloads used in this experiment were generated using IOMeter running on two of the client machines. The workloads accessed two separate logical disks on the same FAStT "6+6" RAID 1/0 array. The BU workload was issuing 32 64 KB sequential reads and then waiting for a *think time* of 30 ms after their completion. The TP workload was generating a purely random 2 KB read/write mix (67% writes), with decreasing think time between I/Os as the run progressed. The workload specifications used in this section were derived from Minerva's performance study [1].

It is, of course, important to apply a more demanding test in which workloads are bursty and highly variable, and in which the results are compared with the system's quantitative objectives. For the next experiment, the test apparatus remained the same (two clients, same back-end). Our two workloads were a decision-support service (DSS) application with relatively large reads (28 KB, 98% reads) and high sequentiality (98%)[1], and a file system server (FS) with many more random operations (62% sequential) and writes (64% writes, 20 KB each). Load intensity was varied by changing the think time between I/O bursts. Think times between 0 ms and 742 ms were used for DSS and between 0 ms and 484 ms for FS. Each think time value and burst pattern was sustained for 30 seconds. For each workload, a fixed sequence of 40 think-time values was used for each 1200-second test run, to apply the same variation of load intensity to a configuration with different SLEDS parameters.

[1] In IOMeter, x% sequentiality x means that each I/O has an $(100-x)$% chance of beginning a new run, that is, starting at a new address range.



Transaction Processing (TP)



Backup (BU)

Figure 5. Competition between a simulated transaction workload (TP) and simulated backup workload (BU) with and without workload control. Response time is shown for both streams, and throughput is measured as hundreds of IOPS. Horizontal lines represent the QoS targets; delivery is successful if either response time is below the target or throughput is above.

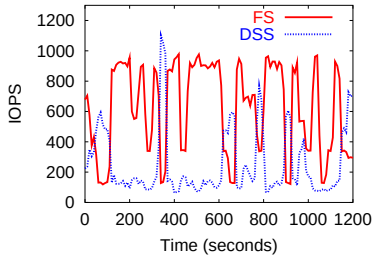


Figure 6. Throughput variations over time of DSS and FS request streams with no throttling applied. DSS throughput decreases when FS increases, because of competition.

6.3.1. Unregulated competition

When SLEDS is set to perform no throttling, the results are as seen in Figures 6 and 7(a). Figure 6 shows the workload variation with time when no throttling is applied. Resource competition between the two streams is clear: one increases when the other decreases. The cause and effect are more readily analyzed by comparing IOPS and response time for each stream (Figure 7(a)). For the FS stream, response time increases with throughput. The DSS throughput, on the other hand, decreases when response time increases, because (as it happens) the FS load has increased.

This competition is neither bad nor good until we have additional information (or assumptions) about system goals. If DSS were a background task then the system would already be operating well. We assume the more interesting case that DSS is an important task with a deadline to perform a fixed set of work–say, for reference, 360,000 I/Os during the 1200-second run. It turns out that SLO targets of 380 IOPS and 150 ms for DSS correlate well with achieving that goal[2]. When there is no throttling, the DSS task fails by 10% to meet its 360k target, and the system fails to deliver against this SLO for most of the sample intervals (62%). These failed intervals appear in Figure 7(a) as points in the *failure region* in the upper left quadrant.

To achieve the goals it is necessary that workload control be applied. The success of SLEDS in achieving the DSS SLO with different operating modes and different requirements for the FS stream is measured and discussed in the following sections. We consider different levels of service that FS might require, expressed as its IOPS limits (manually applied or in its SLO). As a margin for fluctuations, throttle limits are set 5% above the IOPS limits by which SLO success is evaluated.

6.3.2. Fixed throttling limits

One strategy is to allocate somewhat less than half of the total IOPS capacity to each stream, setting a fixed limit of 400 IOPSon each. In SLEDS this is done by applying throttle values manually. The result (Figure 7(b)) is that effective isolation between the streams is achieved, in that each stream's response time is a fairly predictable function of its own throughput.

The nine outlying points with higher response time are artifacts of averaging over a 10-second interval, which we label *interpolated failures*: a transition between successful (throttled) operation at heavy demand and successful operation at light demand or zero demand yields an averaged point for the interval in the failure region, between endpoints in the success region. It may be argued that such points should not be counted as delivery failures, and that the accounting mechanism should be adjusted accordingly, but for this paper they are called failures.

[2]The selection of appropriate SLOs to meet end-to-end requirements is itself a challenging task, beyond the scope of this paper.

Mode & limit	QoS Met		Total ops		
	FS	DSS	FS	DSS	Cumul.
no throttle	100%	37%	757k	327k	1085k
manual (200,400)	85%	96%	230k	379k	611k
manual (400,400)	89%	75%	408k	357k	766k
manual (600,400)	100%	40%	558k	334k	894k
auto (200,400)	98%	99%	337k	586k	924k
auto (400,400)	96%	92%	473k	479k	952k
auto (600,400)	93%	85%	515k	461k	977k

Table 2. Results summary for different throttle settings for DSS and FS request streams. QoS-met columns express fraction of 10-second intervals for which throughput target is exceeded or response time target is met. Throughput target is 95% of throttle limit. Response time target is 120 ms for FS and 150 ms for DSS. The total operations columns show the total number of I/Os over the whole run. Values are shown for the two workloads and the cumulative (Cumul.) throughput.

The overall success of this mode (and of the other modes discussed below) is shown in Table 2. The DSS SLO is achieved 75% of the time and the task falls short of the overall throughput target by only 1%. More aggressive throttling of the FS stream, or a lighter throttle on the DSS stream, is enough for DSS success.

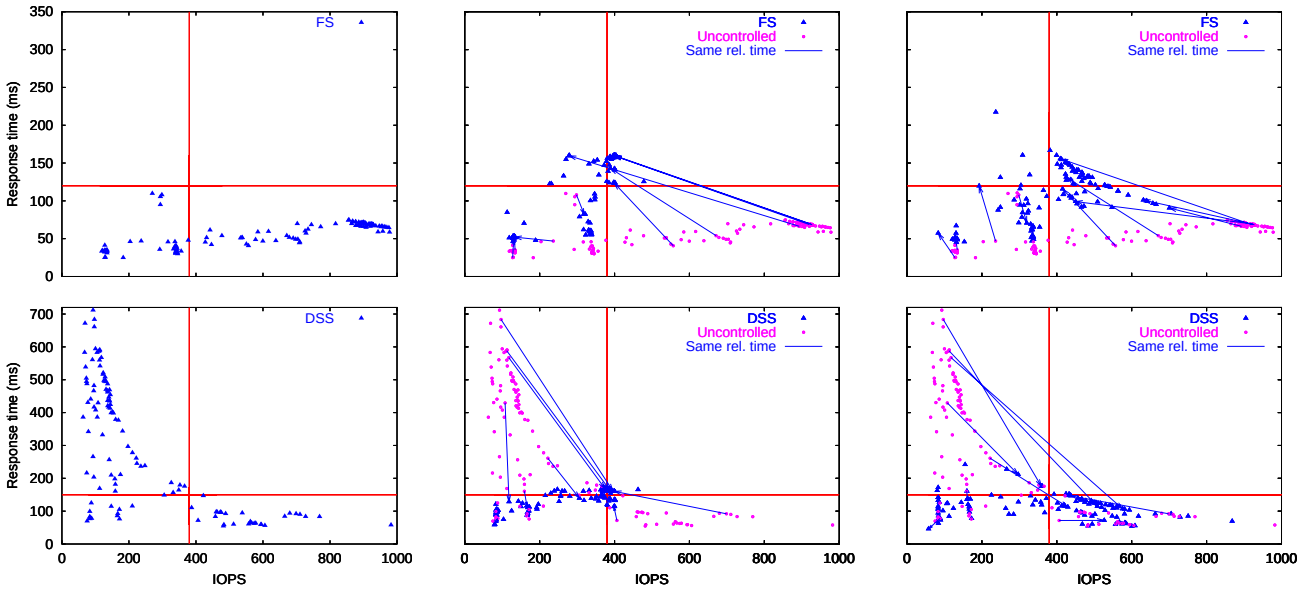
With FS throttled to 200 IOPS, the DSS task succeeds. The only data points that do not meet the SLOs are interpolated failures.

Fixed throughput limits may sometimes be desirable for business reasons. A service provider may choose to give a customer only the throughput that has been paid for. In a typical data center, however, available capacity should be used, provided the use does not interfere with explicit objectives. A fixed-throttle strategy restricts flow even when the unthrottled demand could be handled successfully, so total work by the system is reduced. This drawback is most evident in the decline of the FS total throughput in the fixed-throttle section of Table 2. To solve this problem, SLEDS adjusts throttle values automatically according to load on the system.

6.3.3. Automatic throttling

Automatic adjustment of throttling is guided by the SLOs. The throughput targets determine the most stringent throttling limits, and a comparison of actual response times with targets determines when throttling limits need to be adjusted.

In Figures 7(c) and 8 we see the results from operating with SLOs that match and overcommit the system's capabilities. The "matched" SLOs (Figure 7(c)) have targets of 380 IOPS for both streams. By almost all criteria, the results are



(a) No throttle (b) Manual throttle (c) Automatic throttle

Figure 7. Effect of manual and automatic throttling on QoS for DSS and FS request streams, shown as response time vs. throughput. Each point shows IOPS and response time averaged over a 10-second interval. Lines demarcate the SLOs proposed for these workloads, and points above and to the left are failure points. 7(a). Uncontrolled operation (no throttling). 7(b). Effect of fixed throttle limits of 400 IOPS. 7(c). Automatic throttling governed by SLOs of (380 IOPS, 120 ms) for FS and (380 IOPS, 150 ms) for DSS. In 7(b) and 7(c), the data for uncontrolled operation is superimposed for comparison. Some before-after pairs of corresponding points (same relative time in the run) are connected to show the effect of throttling on particular operating points.

better than for manual throttling at the corresponding limits of 400 IOPS (Figure 7(b)). The DSS throughput increases by 33%, which is comparable to finishing the DSS task in 3/4 the time. The FS throughput is also 16% higher because it is throttled less stringently most of the time. The lighter application of throttling also reduces the total number of QoS failures. However, in this case not all QoS failures are interpolated failures. There are several points which are *response failures* because throttling is not turned on fast enough against a competing workload that has abruptly increased. Occasional failures of this sort are to be expected with strategy of self-adjustment. This is one reason the SLOs for SLEDS do not aim for 100% success, since such a guarantee would require a severe under-use of resources.

The "overcommitted" SLOs (Figure 8) have a target of 570 IOPS for FS. In this scenario, the QoS failure rates are significant (15% for DSS and 7% for FS). Some of these points appear to be steady-state failures: a FS demand that is heavy but below its SLO target can consume enough resource to prevent DSS from meeting its targets. This is no surprise since the sum of targets (380+570=950) is close to the system capacity. Nevertheless, these failures are infrequent enough that the integrated throughput for DSS comfortably exceeds its success criteria, by more than in all the fixed-throttling cases.

7. Conclusions

Consolidation of storage from multiple sites or customers can potentially lead to significant improvements in hardware cost, reliability, and utilization. The downside is that consolidation introduces application failures caused by newly-introduced contention between previously unrelated workloads. Systems also become more complex, exceeding the capabilities of typical manual administration procedures, largely based on rules of thumb.

SLEDS satisfies each I/O stream's performance independently of the requirements of other streams. By doing so, it provides statistically guaranteed performance to applications, thus preventing failures caused by exogenous reasons. We showed how this concept can be efficiently implemented in a largely non-intrusive, totally transparent way on a realistic, enterprise-scale storage system. Our prototype provides exceptional performance isolation properties to client

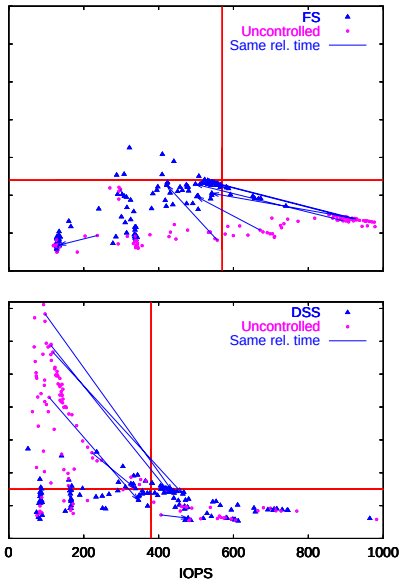


Figure 8. QoS for DSS and FS streams with automatic throttling governed by "overcommitted" SLOs of (570 IOPS, 120 ms) for FS and (380 IOPS, 150 ms) for DSS.

applications, to a level unattainable by stock, best-effort operating systems, protocols, and scheduling policies—while consuming a negligible amount of resources. The resulting system has the best of the consolidated and the non-consolidated worlds: applications no longer suffer performance failures due to resource contention, while they benefit from the (typically much better) reliability of high-end storage devices.

Future work in this area includes better heuristics for automatic throttling. Admission control would require SLEDS to dynamically determine whether sufficient resources are present to service the newcomers. Also, the approach followed here could be extended to provide not only performance, but also dependability guarantees; e.g., keeping multiple redundant paths with adequate bandwidth to a critical subset of the data will result in a substantial availability improvement.

Acknowledgements: We thank Joseph Glider, Young Jin Nam, John Palmer, Archana Samtani, Lee Sanders, Alla Tolchinsky, Chuck Tribolet, Mong-Hang Vo, and Anna Zacchi for their valuable help and insights.

References

[1] G. Alvarez, E. Borowsky, S. Go, T. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, Nov. 2001.

[2] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 175–188, Jan. 2002.

[3] ANSI. SCSI architecture model - 2 (SAM-2), Sept. 2002. Draft Standard, Project 1157-D, Revision 24.

[4] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems*, 6(3):138–151, 1998.

[5] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. *IETF RFC 2475*, 1998.

[6] Brocade Corp. Advanced performance monitoring. White paper. http://www.brocade.com/products/pdf/APM_FNL_101.pdf, 2003.

[7] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk shadowing. In *Proceedings of International Conference on Multimedia Computing and Systems (ICMCS)*. IEEE Computer Society, June 1999.

[8] T. Clark. Zoning for Fibre Channel optics. Fibre Channel Industry Association (Europe) white paper. http://data.fibrechannel-europe.com/technology/white papers/wp57.html.

[9] DataCore Software. SANSymphony version 5 datasheet. White paper. http://www.datacore.com, 2002.

[10] FalconStor Software. Ipstor: build an end-to-end IP-based network storage infrastructure. White paper. http://www.falconstor.com, 2001.

[11] J. Gemmell, H. Vin, D. Kandlur, V. Rangan, and L. Rowe. Multimedia storage servers: A tutorial. *IEEE Computer*, 28(5):40–49, 1995.

[12] IBM. *TotalStorage Virtualization Family SAN Volume Controller—Planning Guide*, July 2003. Pub. GA22-1052-00.

[13] InterSAN Inc. PATHLINE Version 2.5: Delivering storage service excellence. Product data sheet, 2003.

[14] C. Lumb, A. Merchant, and G. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of Conference on File and Storage Technologies (FAST)*, pages 131–144, Apr. 2003.

[15] Open Source Development Lab. Iometer I/O performance analysis tool, Feb. 2003.

[16] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. *Real Time Systems*, 22(1–2):9–48, Jan. 2002.

[17] J. Turner. New directions in communications. *IEEE Communications*, 24(10):8–15, Oct. 1986.

[18] J. Wilkes. Travelling to Rome: QoS specifications for automated storage system management. In D. H. L. Wolf and R. Steinmetz, editors, *Proceedings of 9th International Workshop on Quality Of Service (IWQoS)*, pages 75–91. Springer Verlag, June 1991.