Hierarchical Disk Sharing for Multimedia Systems

Joel C. Wu
jwu@cs.ucsc.edu

Scott Banachowski
sbanacho@cs.ucsc.edu

Scott A. Brandt
sbrandt@cs.ucsc.edu

Computer Science Department
University of California, Santa Cruz
Santa Cruz, CA 95064

ABSTRACT

Systems that use or serve multimedia data require timely access to data on hard drives. To ensure adequate performance users must either prevent overload of disk resources, or use real-time algorithms that rely on intricate knowledge of disk internals to meet deadline requirements. We have developed Hierarchical Disk Sharing (HDS) to allow disks to be fully utilized while sustaining a bandwidth reservation, without requiring detailed knowledge of the drive internals. HDS uses a hierarchy of token bucket filters to isolate disk access among clients and groups of clients, and to allow for reclaiming of unused bandwidth. We discuss the design of HDS and present our implementation in a Linux block device driver, demonstrating the effectiveness (and limitations) of this approach.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*; D.4.8 [**Operating Systems**]: Performance

General Terms

Algorithms, Design, Performance

Keywords

Storage, Disk, QoS, Multimedia, Soft Real-Time

1. INTRODUCTION

Multimedia computing is ubiquitous and it is common to find systems concurrently supporting both multimedia and general-purpose workloads. This is also true for storage, as multimedia files reside on the same storage devices as other data. To meet their timing constraints, multimedia applications must access stored data in a timely manner or experience undesired performance such as jitter, drop-outs, or other audio and video glitches.

Applications typically access data without regard for others and may successively request multiple blocks as fast as the storage device permits. This may negatively impact performance when multimedia applications, which require sustained disk bandwidth to read and decode data on time, must compete for disk resources while other applications use it for non-real-time purposes. Since we cannot in general expect applications to cooperatively share the disk, it is the role of the operating system to provide resource isolation by allowing time-critical applications to access data at reserved levels of service (*reservations*), or by limiting the amount that competing applications may interfere with each other (*fairness*).

This paper describes Hierarchical Disk Sharing (HDS), an algorithm for reserving and partitioning the bandwidth to disk and other block storage devices among different applications, or groups of applications (which we call *classes*). The principle behind HDS is based on the hierarchical link sharing for networks [6]. HDS provides capability that is absent in current commodity operating systems. One of our main design goals is that the reservation mechanism be independent of high-level features like file-systems, and low-level features like disk schedulers, so that it can be employed across many systems, including storage network devices. Therefore we chose to implement our prototype of HDS in the block device layer of the Linux kernel.

2. BACKGROUND AND RELATED WORK

Time-sensitive storage access is addressed by real-time disk scheduling algorithms [7]. Generally, real-time disk schedulers are designed for homogeneous applications such as video-on-demand. In such systems, disk requests have deadlines. The requests are typically ordered based on some combination of real-time scheduling technique and disk seek optimization scheme (*e.g.* SCAN-EDF [8]). These algorithms do not support mixed-workload environment well.

More elaborate disk schedulers (whether real-time or not), which take into account the seek time and rotational latency, require the use of accurate disk models. Using such low-level information may provide predictable service guarantees [9]. A challenge facing the practical deployment of these schedulers in commodity systems lies in their dependency on intricate knowledge of drive internals (which must be determined by probing the disks [5, 13]). This run-time information is difficult to obtain for an external disk scheduler, and low-level disk properties must be known and well-characterized for each different model of drive. Exacerbating the issue is the increasing intelligence of drive firmware, which limits the effectiveness of these schedulers to special-purpose systems. Our goal is to provide better support using com-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
NOSSDAV'05, June 13–14, 2005, Stevenson, Washington, USA.
Copyright 2005 ACM 1-58113-987-X/05/0006 ...$5.00.

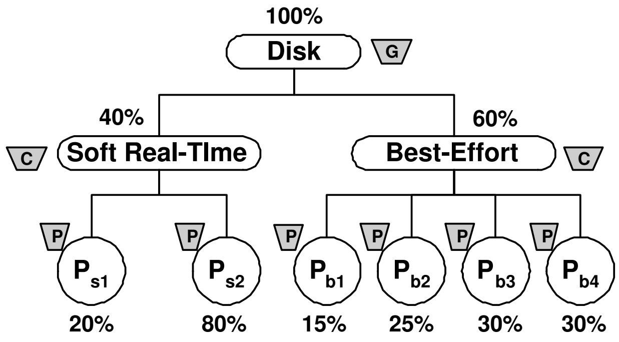


Figure 1: HDS allows arbitrary mix of shares controlled by Global bucket, Class buckets, and Process buckets.

modity equipment and enable shaping of bandwidth without relying on the disk scheduler.

Real-time disk schedulers provide fine-grained control of disk requests based on the deadlines of the disk requests. For many applications, this amount of control is unnecessary. For example, for many soft real-time applications it is more important to achieve a constant data rate (a constant quality of service (QoS)) than to meet the deadlines of individual disk requests. Our approach is to provide better QoS over aggregate data rates.

Some QoS-aware schedulers are designed specifically to address mixed workloads [4, 11, 12]. These are typically two-level schedulers that classify disk requests into categories such as real-time, best-effort, and interactive. Each class has its own scheduler, and the requests from different classes are merged by a meta-scheduler. The number of classes is limited and static. Our multi-level approach allows the dynamic creation and removal of classes at run-time. YFQ [1] offers similar capability to HDS, using different mechanisms.

The techniques we employ in HDS are already used for managing networking resources. Our resource allocation policies are based on the hierarchical link-sharing mechanism for networks [6]. Like that system, we isolate traffic by separating the requests of different applications or types of application; disk blocks are analogous to packets, and bandwidth represents the number of blocks accessed over a period of time. The mechanism we use to control disk bandwidth is also based on a networking technique: token bucket filters [10], which were designed for network flow control. However, unlike network bandwidth, the available disk bandwidth fluctuates due to variations in seek times for different disk requests, complicating its management.

3. HIERARCHICAL DISK SHARING

Traditional disk access is best-effort, and there are no timing guarantees. Acceptable performance is achieved when the disk is not overloaded. When demand for disk bandwidth exceeds the supply, all of its users may experience performance degradation.

Our approach to this problem is to provide a mechanism that allows reservations of disk bandwidth, graceful degradation under heavy load, and reclaiming of unused reservations. A hierarchical structure for resource sharing provides a basis for meeting all of these goals. This section describes the Hierarchical Disk Sharing (HDS) approach.

3.1 Disk usage accounting

Specifying a disk reservation by bandwidth is intuitive, but disk bandwidth is not constant; service times vary depending upon the initial position of the read-write head, the position of the requested data on the disk, the low-level disk scheduling algorithm, *etc.* Translating a bandwidth requirement into low-level disk operations is a complicated task [2]. Alternatively, specifying a disk reservation by a reserved time-slice (instead of bandwidth) may result in different amounts of data being retrieved per time-slice, and the specification of disk time is not intuitive from the user's perspective.

Existing reservation-capable schedulers must contend with this issue. The Cello [11] scheduler presents two methods of accounting, either by size or time. Some schedulers allow reservations in terms of number of requests [1, 12]. However, requests may also vary in size and service time.

To fulfill QoS goals, the system must provide performance in line with the users' expectations, regardless of the amount of work that the disk is actually doing on behalf of different users. We expect users to perceive the quality of service for disk by the bandwidth (data rate) that it can provide. Therefore HDS accounts for disk usage in terms of bandwidth and does its accounting based on the actual amount of data transferred.

3.2 Hierarchical structure

In HDS, a disk's bandwidth is divided between applications in a hierarchical tree structure; an example is pictured in Figure 1. Each leaf node represents a point of control for accessing the disk. Therefore, each leaf-node is associated with an individual client of the disk; in our implementation a leaf node corresponds to a Linux process. When a process first attempts to access the disk, a leaf node is created and added to the tree. When it quits, its node is removed.

Non-leaf nodes are called *classes*, and represent a group of clients. The children of a class node may be leaf nodes or other class nodes. For example, a class node might represent a user logged onto the system, with all its children representing processes created by this user. The parent of the user class node may itself be a class node representing the user's department. Figure 1 demonstrates using classes to isolate best-effort and real-time processing, an approach useful for multimedia servers where the requests with time constraints should be isolated from other traffic.

Hierarchies may be created to control disk accesses in many flexible configurations, and we expect that a system administrator or other expert may want to choose how to set up a particular system. Our system has an interface for constructing the desired class hierarchy, including dynamically adding and deleting classes; for brevity the interface description has been left out of this paper.

3.3 Hierarchical reservation

Each node x has an associated reservation r_x, determined by the reservations of nodes above it in the tree structure. There are two modes for a node to specify its reservation: either an *absolute* fraction f_x, or a *relative* fraction, based on a weight w_x, of the parent node's reservation. The root node has an absolute fraction of 1.

If a node x has an absolute reservation f_x (between 0 and 1), its reservation is this fraction of its parent's reservation. For example, because the root node has $r_0 = 1$, a child of the root with $f_x = 0.4$ will have a reservation of $r_x = f_x r_0 = 0.4$. The sum of absolute fractions among any node's children may not exceed 1.

A class's bandwidth that is not used by absolute reservations is shared by its other children in proportion to their relative weights. For example, consider a class node (node p) with reservation $r_p = 0.5$, and three children nodes (a,b, and c), the first with absolute fraction $f_a = 0.4$, and the others with relative weights $w_b = 4$ and $w_c = 6$. Because node a is absolute $r_a = 0.4r_p = 0.2$, leaving $0.6r_p = 0.3$ left for nodes b and c. Node b's reservation is $r_b = (0.6r_p)4/(4+6) = 0.24r_p = 0.12$, and node c's reservation is $r_c = (0.6r_p)6/(4+6) = 0.36r_p = 0.18$.

In the default setting, clients are added to a parent node with equal weight to promote fair sharing. When a client needs a higher level of service than others, it may do so by either increasing its weight or requesting an absolute fraction of its parent bandwidth. If the nodes on the path from a client to the root all have absolute reservations, then the client effectively reserves a static fraction of the total disk bandwidth; if any node in this path has a relative reservation, the node's reservation may vary when other nodes join or leave the structure. HDS allows administrators to set permissions for adding classes or nodes, changing reservations, admission control, etc. These policy-based controls are outside the scope of this paper.

3.4 Token bucket implementation

In the previous section we explained that each node in the hierarchical tree structure has a value representing its current reservation of the disk. We now describe our mechanism for enforcing these reservations using token bucket filters at each node of the hierarchy. Current Linux distributions include a Hierarchical Token Bucket (HTB) filter for the network driver [3]. We developed HDS for the block device driver independently of the network driver— although the basic goals are essentially the same, they share no common code and the algorithms differ.

Disk bandwidth may be controlled at different points in the I/O stack. HDS is at the block-device layer, which is below the file system and above the disk scheduler. The regulation of disk bandwidth in HDS is implemented using token bucket filters. In order to make disk requests, a client must possess tokens. In HDS, each token represents 1 KB of data, meaning a request for 16 KB of data requires 16 tokens. Each node x in the hierarchy has an associated bucket, which may hold up to N_x tokens. When a client request is serviced, tokens are removed from its bucket. Tokens are replenished at a rate corresponding to the client's reservation. If the root token rate is T_0, then its child node x with reservation r_x will replenish tokens at rate $T_x = r_x T_0$. The root token rate represents the entire bandwidth of a disk. For example, if a disk supports an average throughput of 20 MB/second, the root token rate $T_0 = 20K$ tokens/second.

Although every node has a token bucket, only leaf nodes make requests. The token buckets of non-root nodes facilitate sharing of bandwidth. In addition to its own tokens, a node may use tokens from its parent (which in turn may use those of its parent). The effect is that unused bandwidth is shared first among nodes of the same class, then among parent class, and eventually, globally.

```
replenish($x$)
  $c_x = \max(c_x + (t - l_x)T_x,\ N_x)$
  $l_x = t$

decrement($x, size$)
  $c_x = c_x - size$
  if($P_x != $ root) decrement($P_x, size$)

node can_make_request($x, size$)
  if($size > c_x$) replenish($x$)
  if($c_x \geq size$) return $x$
  if($P_x != $ root) return can_make_request($P_x, size$)
  return null

make_request($x, size$)
  $y$ = can_make_request($x, size$)
  if($y$)
      decrement($y, size$)
  else
     decrement($x, size$)
     sleepfor(($size - c_x)/T_x$)
  endif
  do disk access
```

Figure 2: Functions used by the HDS algorithm.

Definitions. The following definitions are used to describe the algorithm: The current time is t. For a node x, P_x is its parent, N_x is its bucket size, c_x is the number of tokens currently in its bucket, T_x is the token rate, and l_x is the last time its bucket was replenished. The pseudo-code for the algorithm is in Figure 2.

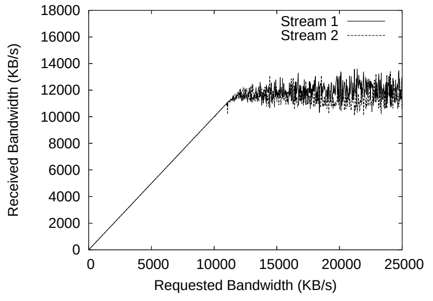
To make a disk request, the client calls the function `make_request()`, with arguments indicating its node and the size of the request (all client nodes are leaf nodes). If the client, or any of its ancestors, has enough tokens to satisfy the request (determined by `can_make_request()`), then the disk access proceeds immediately. Otherwise, the calling process is suspended until enough time passes to acquire the necessary tokens. This blocking mechanism forces clients to throttle their request rate to their token rate, thus enforcing the bandwidth reservation.

The function `can_make_request()` determines if a node has enough tokens to meet the request. If the node's current token supply is low, it calculates the number of tokens that should be replenished since its last request. If the node still does not have enough tokens to service the request, it determines if a higher-level node in its hierarchy has enough by recursively calling `can_make_request()` on its parent. If no node in the caller's path to the root may satisfy the request, `can_make_request()` returns *null*, otherwise it returns the identity of the node that satisfies the request.

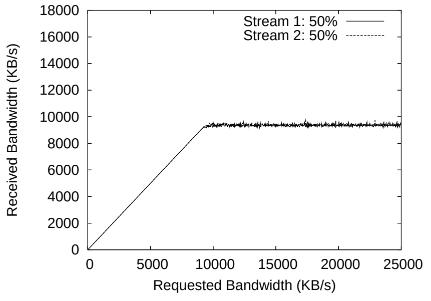
Before servicing the disk request, `make_request()` decrements the tokens from the bucket of the node supplying the tokens. Decrement is a recursive function, so tokens are also removed from the buckets of all higher-level nodes.

3.5 Properties

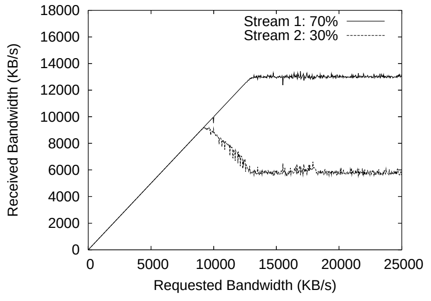
Only leaf nodes initiate requests. Because every node has its own supply of tokens, HDS guarantees that every node is always able to freely request up to its reservation in disk requests (allocated at rate T_x, and corresponding to x's reservation r_x). HDS also allows clients to consume any extra bandwidth by using tokens supplied by parent nodes



(a) Normal Linux system behavior.



(b) With HDS sharing of 50%-50%.



(c) With HDS sharing of 70%-30%.

Figure 3: The effect of overload on throughput.

recursively, up to the root node. This hierarchical structure provides a method for clients to reclaim bandwidth that is reserved, but not used, by others.

When a node drains tokens from its own bucket, it also drains those from higher-order buckets. The result is that when a class's children make disk requests, the class's tokens will be drained as well. If some of the children are not fully using their reservation, the parent will have surplus tokens. These tokens are available to other children when their own supply runs out, so that a node that has exceeded its reservation may still be able to proceed. Bandwidth isolation is preserved not only between leaf nodes, but at the class level and, in fact, at every level of the hierarchy.

4. EXPERIMENTS WITH HDS

We ran several experiments to demonstrate the ability of HDS to shape disk traffic, using synthetic applications to generate disk workload. We focus mostly on read workloads both because multimedia is typically read intensive and because write performance is often aided by buffering. Our test system is a 1.5 GHz P4 with 512MB of RAM. The disk is a Seagate ST340810A IDE drive formatted with the ext2 file system.

Figure 3 shows a situation where disk bandwidth has become saturated. Two processes are reading from the disk simultaneously, at the same rate. The x-axis shows the requested bandwidth and the y-axis shows the measured received bandwidth. Figure 3(a) shows the result on unmodified Linux. Both processes receive their desired bandwidth until the disk became saturated with requests (*i.e.* the sum of requested bandwidth exceeds the total disk bandwidth). There is no isolation, so at that point actual throughput is unpredictable and varies considerably.

HDS provides reservation and isolation of bandwidth. Figure 3(b) shows the same experiment with HDS, where each task reserves equal relative weight. At saturation the bandwidth divides evenly between the streams and the achieved throughput is very stable. This fair-sharing comes at the expense of slightly lower overall disk throughput because we limit the number of requests. Section 5 discusses future work for alleviating this effect by varying the root token rate based on observed performance.

Figure 3(c) shows the same workload again, while demonstrating HDS's reservation capability. We allocated 70% of the disk to stream 1 and 30% to stream 2. Below disk overload, each process receives what they request. As the requested bandwidths exceed the disk capability, the reservation mechanism begins to take effect. Stream 1 continues to receive a higher level of service based on its share. Once the 70%-30% share is reached, the bandwidth received by each process remains constant as the rate of disk requests increases. The average throughput for Stream 1 at full rate is 12.7 MB/s, about 70% of the total disk bandwidth of 18 MB/s , and for Stream 2 it is 5.7 MB/s, about 30% of the total bandwidth.

The next experiment shows the ability of HDS to provide hierarchical resource sharing. We created two classes, A and B, each reserving 50% of the disk. Stream 1 belongs to Class A, so it reserves 100% of the class reservation. Streams 2 and 3 belong to Class B, and reserve 65% and 35% of Class B's reservation, respectively. Figure 4 shows that all three streams receive bandwidth corresponding to their allocation, with no interference from each other.

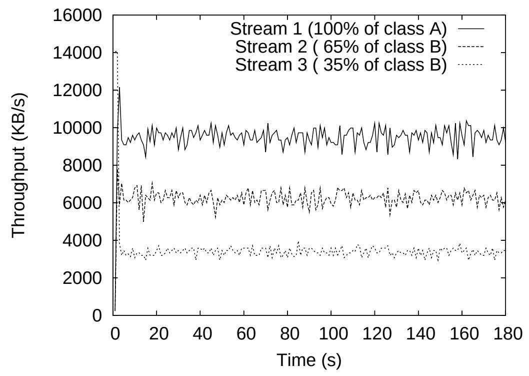


Figure 4: Isolation of bandwidths. Class A and B reserve 50% each. All streams are greedy.

Excess bandwidth may be available when a process needs more than its reserved share. Figure 5 shows this scenario. In this experiment, there are two classes and three streams. Class A and B each reserve 50%. Stream 1 and 2 belong to Class A and reserve 80% and 20% of its bandwidth. Stream 3 belongs to class B, so receives 100% of its bandwidth. At the beginning, only Stream 1 is active. Although its total share is only a fraction of the total bandwidth (its share is 40%), because no other tasks are active it receives the total disk bandwidth. At time 60 Stream 2 becomes active. There is still excess bandwidth because Class A's share is only 50%. The excess bandwidth is distributed to Streams 1 and 2. From time 60 to time 120, they receive

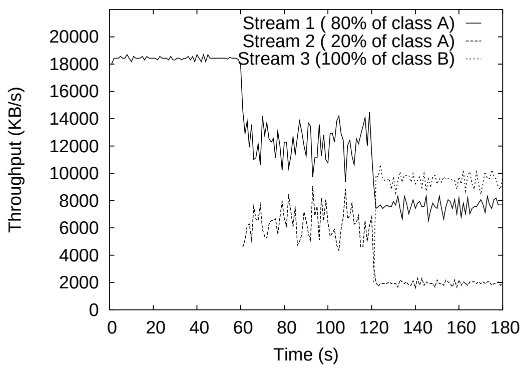


Figure 5: Using unassigned bandwidth. Class A and B reserve 50% each. All streams are greedy.

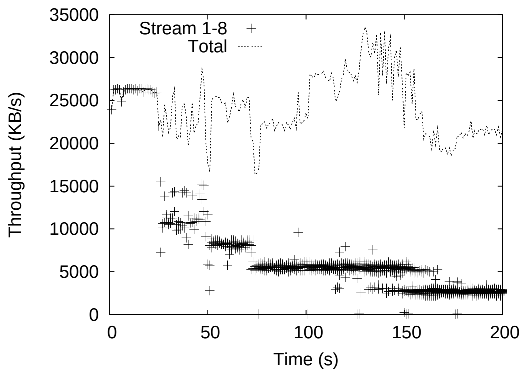


Figure 6: Effect of changing workload on total throughput.

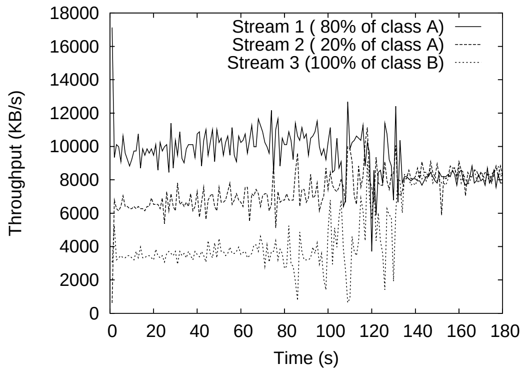


Figure 7: Effect of increasing root node bandwidth on sharing.

their fair-share plus the excess bandwidth. Excess bandwidth is allocated on first-come first-serve basis, accounting for the observed variation in actual rate (this variation is a topic for future investigation). At time 120 Stream 3 begins and, now fully loaded, the nominal reservations are enforced.

5. FUTURE WORK

The throughput of a disk during overload depends on the workload. Since the totality of resource may vary, proportional sharing is often used to allocate a fraction of the bandwidth to different shares [11, 1]. However, because the total disk bandwidth varies, these fractional guarantees are not constant. In many circumstances it would be more useful to have assurances in terms of data rates, and not in terms of a fixed fraction of a dynamically changing total bandwidth.

HDS is able to provide assurance in terms of data rates with *absolute* reservation and knowledge of the root node bandwidth. However, the unpredictable nature of storage also presents a problem for HDS. Thus far we have assumed that the disk bandwidth is fixed. Although convenient, as already noted, this assumption is not true; the throughput of a disk during overload depends on the workload. This is highlighted in the experiment shown in Figure 6 (without HDS), where we introduced 8 greedy streams in 25 second intervals. The upper line shows total throughput. As the offered load increases, the total achieved bandwidth varies significantly.

HDS uses a conservative estimate of total bandwidth. This under-utilizes the disk, but allows HDS to provide the reserved bandwidth. The more aggressively HDS utilizes the disk, the less effective it is at providing isolation and sharing. In Figure 7 (which has the same workload as Figure 4), we show what happens as we increase the bandwidth of the root node by 384 KB/s every 20 seconds. In this case, the share mechanism becomes less effective, failing completely about 2 minutes into the experiment, after a 2 MB/s increase in disk bandwidth.

We plan to increase the performance of HDS by addressing this issue. One possible solution is to adapt the total rate of the root node based on the current disk performance or the effectiveness of the current reservations. For example, we may monitor the actual rates at nodes during overload and, if they do not match their reservations, we may adjust the root token rate downward until they do.

We implemented a heuristic to show the benefit of adjusting the root node's bandwidth in response to performance, shown in Figure 8. Stream 2 begins at time 40, and the two streams maintain their relative share of 3:1 from time 40 to 70. The adaptive mechanism is triggered at time 70; it attempts to increase the total bandwidth without violating the reservations. The algorithm increases the global token rate if the current measured shares are accurate; otherwise it decreases the global token rate. After the adjustment at time 70, the throughput of Stream 1 is increased while Stream 2 maintains its constant rate. The algorithm is slow to adapt, but demonstrates the potential of this approach.

With a dynamically changing bandwidth for the root node, the *absolute* and *relative* weighted sharing of HDS can be augmented by a third type of reservation, *absolute bandwidth* reservation, where a portion of the bandwidth is set aside for reservation in terms of bandwidth, and the rest of the bandwidth is available for proportional reservation by either *absolute* or *relative* weighted sharing. The portion of bandwidth set aside for absolute reservation can be pro-

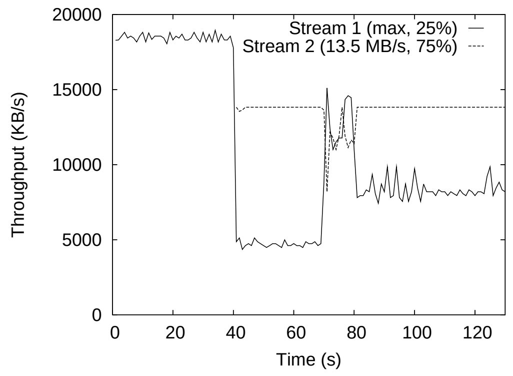


Figure 8: Maximizing total throughput.

tected during overload by monitoring and throttling the rate of the proportional share portion.

Adapting the root token rate dynamically also solves another problem for HDS: the throughput of a disk is unknown without testing. In our experiments, the default token rate was set to a value we predetermined based on previous average throughput measurements. If deployed in a commodity system, the disk performance will be unknown. Using an adaptive root token rate will allow HDS to dynamically discover appropriate rates when applied to an unknown disk.

Other future work includes investigating the interaction between HDS and low-level disk schedulers. Although one of our stated goals is to avoid dependence on the disk scheduler, our algorithm need not ignore that many low-level schedulers have similar goals which we can leverage by managing the ordering of related requests. Another area to investigate is the effect of prefetching on our token-cost structure. In addition to the streaming workloads of file data, the block device driver where we implemented HDS sees requests for metadata and prefetches from the file system, facts that HDS has so far ignored.

6. CONCLUSION

Hierarchical Disk Sharing (HDS) supports flexible and dynamic sharing of storage resources. It allows processes to reserve disk bandwidth and effectively isolates disk clients, providing them with their reserved share. HDS uses a hierarchical structure for allocating disk bandwidth that simplifies reservations and allows processes to reclaim unreserved or reserved but unused bandwidth. We implemented HDS in Linux and demonstrated its ability to isolate the disk traffic of clients and provide storage quality of service. During overload, clients receive at least their reserved share of bandwidth, in isolation from other competing loads. The effectiveness of HDS is determined by the choice of the algorithm's maximum disk bandwidth. We are currently developing an algorithm that will allow the entire disk bandwidth to be dynamically adjusted, automatically adapting to new platforms and changing workloads. Our eventual goal is to apply HDS in an overall end-to-end QoS framework for distributed storage.

Acknowledgments

This research was supported in part by Lawrence Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under contract B520714; also by Intel Corporation and a DOE High-Performance Computer Science Fellowship.

7. REFERENCES

[1] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 400–405, June 1999.

[2] S. Childs. Portable and adaptive specification of disk bandwidth quality of service. In *Proceedings of the 9th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, June 1999.

[3] M. Devera. Hierarchical token bucket. http://luxik.cdi.cz/~devik/qos/htb/.

[4] Z. Dimitrijevic and R. Rangaswami. Quality of service support for real-time storage systems. In *Proceedings of the International IPSI-2003 Conference*, October 2003.

[5] Z. Dimitrijevic, R. Rangaswami, and E. Chang. Diskbench: User-level disk feature extraction tool. Technical report, UCSB, November 2001.

[6] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, 1995.

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

[8] A. L. Reddy and J. Wyllie. Disk scheduling in a multimedia I/O system. In *Proceedings of ACM Conference on Multimedia*, pages 225–233. ACM Press, 1993.

[9] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*. IEEE, December 2003.

[10] S. Shenker, C. Partridge, and R. Guerin. Specification of guaranteed quality of service. *RFC 2212*, September 1997.

[11] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 44–55. ACM Press, 1998.

[12] R. Wijayaratne and A. L. Reddy. Integrated QOS management for disk I/O. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 487–492, June 1999.

[13] B. Worthington, G. Ganger, Y. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 146–156. ACM Press, 1995.