CA-NFS: A Congestion-Aware Network File System

Alexandros Batsakis
NetApp
Johns Hopkins University

Randal Burns
Johns Hopkins University

Arkady Kanevsky
NetApp

James Lentini
NetApp

Thomas Talpey
NetApp

Abstract

We develop a holistic framework for adaptively scheduling asynchronous requests in distributed file systems. The system is holistic in that it manages all resources, including network bandwidth, server I/O, server CPU, and client and server memory utilization. It accelerates, defers, or cancels asynchronous requests in order to improve application-perceived performance directly. We employ congestion pricing via online auctions to coordinate the use of system resources by the file system clients so that they can detect shortages and adapt their resource usage. We implement our modifications in the Congestion-Aware Network File System (CA-NFS), an extension to the ubiquitous network file system (NFS). Our experimental result shows that CA-NFS results in a 20% improvement in execution times when compared with NFS for a variety of workloads.

1 Introduction

Distributed file system clients consume server and network resources without consideration for how their operations interfere with their future requests and other clients. Each client request incurs a cost to the system, expressed in increased load to one or more of its resources. As more capacity, more workload, or more users are added congestion rises, and all client operations share the cost in delayed execution. However, clients remain oblivious to the congestion level of the system resources.

When the system is under congestion, network file servers try to maximize throughput across clients, assuming that their benefit increases with the flow rate. This practice does not correspond well with application-perceived performance because it fails to distinguish the urgency and relative priority of file system operations across the client population. From the server's perspective, all client operations at any given time are equally important. This is a fallacy. File system operations come at different priorities implicitly. While some need to be performed on demand, many can be deferred. Synchronous client operations (metadata, reads) benefit more from timely execution than asynchronous operations (most writes, read-aheads), because the former block the calling application until completion. Also, certain asynchronous operations are more urgent than others depending on the client's state. For example, when a client's memory consumption is high, all of its write operations become synchronous, leading to a degradation in system performance.

In this paper, we develop a performance management framework for distributed file systems that dynamically assesses system load, manages system resources, and schedules asynchronous client operations. When the system resources approach critical capacity, we apply priority scheduling, preferring blocking to non-blocking requests, and priority inheritance, *e.g.* performing writes that block reads at high priority, so that non-time-critical (asynchronous) I/O traffic does not interfere with on-demand (synchronous) requests. On the other hand, if the system load is low, we perform asynchronous operations more aggressively in order to avoid the possibility of performing the same operations at a later time, when the server resources will be congested.

The framework is based on a *holistic* congestion pricing mechanism that incorporates all critical resources among all clients and servers, from client caches to server disk subsystems. Holistic goes beyond end-to-end in that it balances resource usage across multiple clients and servers. (End-to-end also connotes network endpoints and holistic management goes from client applications to server disk systems.) The holistic approach allows the system to address different bottlenecks in different configurations and respond to changing resource limitations over time.

Servers encode their resource constraints by increasing or decreasing the price of asynchronous reads and writes in the system in order to "push back" at clients.

As the server prices increase, the clients that are not resource constrained will defer asynchronous operations for a later time and, thus, reduce their presented load. This helps to avoid congestion in the network and server I/O system caused by non-critical operations.

The underlying pricing algorithm, based on resource utilization, provides a `log-k` competitive solution to resource pricing when compared with an offline algorithm that "knows" all future requests. In contrast to heuristic methods for moving thresholds, this approach is system and workload independent.

We evaluate our proposed changes in CA-NFS (Congestion-Aware Network File System), an extension of the NFS protocol, implemented as modifications to the Linux NFS client, server, and memory manager. Experimental results show that CA-NFS outperforms NFS and improves application-perceived performance by more than 20% in a wide variety of workloads.

2 System Operation

In this section, we give the intuition behind scheduling asynchronous operations and the effect these have on system resource utilization. We then demonstrate how clients adapt their behavior using pricing and auctions.

2.1 Asynchronous Writes

The effectiveness of asynchronous write operations depends on the client's current memory state. Writes are asynchronous only if there is available memory; a system that cannot allocate memory to a write, blocks that write until memory can be freed. This hampers performance severely because all subsequent writes become effectively synchronous. It also has an adverse effect on reads. All pending writes that must be written to storage interfere with concurrent reads, which results in queuing delays at the network and disk.

CA-NFS changes the way that asynchronous writes are performed compared to regular NFS. NFS clients write data to the server's memory immediately upon receiving a `write()` system call and also buffer the write data in local memory. The buffered pages are marked as dirty at both the client and the server. To harden these data to disk, the client sends a `commit` message to the server. The decision of when to commit the data to the server depends on several factors. Traditionally, systems used a periodic update policy in which individual dirty blocks are flushed when their age reaches a predefined limit [32]. Modern systems destage dirty pages when the number of dirty pages in memory exceeds a certain percentage (flushing point), which is typically a small fraction of the available memory (*e.g* 10%). Then, a daemon wakes up and starts flushing dirty pages until an adequate number of pages have reached stable storage.

In contrast to regular NFS, CA-NFS clients adapt their asynchronous write behavior by either *deferring* or *accelerating* a write. CA-NFS clients accelerate writes by forcing the CA-NFS server to sync the data to stable storage so that the client does not need to buffer all of the corresponding dirty pages. The idea behind write acceleration is that if the server resource utilization is low, there is no need to defer the commit to a later time. Also, clients may elect to accelerate writes in order to preserve their cache contents and maintain a high cache hit rate. Note that accelerating a write does not make the write operation synchronous. Instead, it invokes the write-back daemon at the client immediately.

Write acceleration possibly increases the server disk utilization and uses network bandwidth immediately. In write-behind systems, many writes are canceled before they reach the server [5, 34], *e.g.* writing the same file page repeatedly, or creating and deleting a temporary file. Thus, the load imposed to the server as a result of write acceleration could be avoided. However, write acceleration has almost no negative effect on system performance, because CA-NFS accelerates writes only when the server load is low.

Deferring a write avoids copying dirty data to server memory upon receiving a write request. Instead, clients keep data in local memory only, until the price of using the server resources is low. Clients price asynchronous writes based on their ability to cache writes, *i.e.* available memory. A client with scarce memory, because of write deferral, will increase its local price for writes so that its buffered pages will be transferred to the server as soon as possible. To make write deferral possible, we modify the operation of the write-back daemon on the clients by dynamically changing the flushing point value based on the pricing mechanism to dictate when the write-back of dirty pages should begin.

Deferring a write consumes client memory with dirty pages, saves server memory, and delays the consumption of network bandwidth and server disk I/O. However, it faces the risk of imposing higher latency for subsequent synchronous `commit` operations. This is because a file sync may require a network transfer of the dirty buffers from the client to server memory. Note that deferring a write does not guarantee that the server price for the same operation will be lower in the future. Instead, this policy gives priority to operations originating from resource-constrained clients.

CA-NFS follows NFS's close-to-open consistency model. Deferring or accelerating writes does not violate the consistency semantics of NFS, because CA-NFS does not change the semantics of the `COMMIT` operation. Asynchronous write-back in NFS includes a deadline that, when it elapses, escalates the operation to a synchronous write. CA-NFS does the same.

The server prices asynchronous writes based on its memory, disk and network utilization. If the server memory contains blocks that are currently accessed by clients, setting high prices forces clients to defer writes in order to preserve cache contents and maintain a high cache hit rate. Also, if the disk or network resources are heavily utilized, CA-NFS defers writes until the load decreases, to avoid queuing delays because of pending writes that must be written to storage and interfere with concurrent, synchronous reads. If the system resources are underutilized, the server encourages clients to flush their dirty data by decreasing its price.

2.2 Asynchronous Reads

CA-NFS attempts to optimize the scheduling of asynchronous reads (read-ahead). Servers set the price for read-aheads based on the disk and network utilization. If the server resources are heavily congested, CA-NFS servers are less willing to accept read-ahead operations.

A client's willingness to perform read-ahead depends on its available memory and the effectiveness of the operation. If the server and network resources are congested so that the server's read-ahead price is higher than their local price, clients perform read-ahead prudently in favor of synchronous operations. Capping the number of read-ahead operations saves client memory, delays the consumption of network bandwidth, but often converts cache hits into synchronous reads because data were not preloaded into the cache. On the other hand, if the server price is low, clients perform read-ahead more aggressively.

2.3 CA-NFS in Practice

Figure 1 shows the high-level operation of the system and how the pricing model make clients adapt their behavior based on the state of the system. At this time, our treatment of pricing is qualitative. We describe the details of constructing appropriate pricing models in Section 3.3.

The server sets the price of different operations to manage its resources and network utilization in a coordinated fashion. In this example, the server's memory is near occupancy and it is near its maximum rate of I/O per second (IOPS). Based on this, it sets the price of asynchronous writes to be relatively high, because they consume server memory and add IOPS to the system.

CA-NFS allows the system to exchange memory consumption between the clients and the server. Clients adapt their prices based on their local state. Client #1 has available memory, so it stops writing dirty data. Client #2 is nearing its memory bound and, if it runs out of memory, applications will block awaiting the completion of asynchronous writes. Thus, even though the server price of asynchronous writes is high, this client is willing to pay in order to avoid exhausting its memory. When the server clears its memory, it will lower the price of asynchronous writes and Client #1 will commence writing again. Servers notify clients about their prices as part of the CA-NFS protocol.

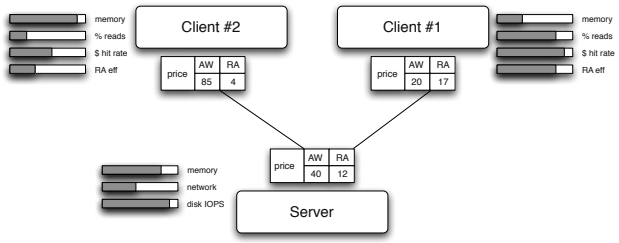


Figure 1: Overview of Congestion-Aware NFS. Clients and servers monitor their resource usage from which they derive prices for the different file system operations. (AW = asynchronous write, RA = read ahead, RA eff = read-ahead effectiveness.)

The criteria for whether to perform read-ahead prudently or aggressively are similar. Client #1 has lots of available memory, a read-dominated workload, and good read-ahead effectiveness, so that read-ahead turns most future synchronous reads into cache hits. Thus, it is willing to pay the server's price and perform more aggressive read-ahead. Client #2 has a write-dominated workload, little memory, and a relatively ineffective cache. Thus, it halts read-ahead requests to conserve resources for other tasks.

3 Pricing Mechanism

In distributed file systems, resources are heterogeneous and, therefore, no two of them are directly comparable. One cannot balance CPU cycles against memory utilization or vice versa. Nor does either resource convert naturally into network bandwidth. This makes the assessment of the load on a distributed system difficult. Previous models [20, 38, 44] designed to manage load and avoid throughput crashes via adaptive scheduling focus on one resource only or rely on high-level observations, such as request latency. The price unification model in CA-NFS provides several advantages: (a) it takes into account all system resources, (b) it unifies congestion across all devices in order to be comparable, and (c) it identifies bottlenecks across all clients and the server in a collective way.

Underlying the entire system, we develop a unified algorithmic framework based on competitive analysis for the efficient scheduling of distributed file system operations with respect to system resources. We rely on the algorithm of Awerbuch *et al.* [4] for bandwidth sharing in circuit-sharing networks with permanent connec-

tions that uses an online auction model to price congestion in a resource independent way. We adapt this theory to distributed file systems by considering the path of file system operations, from the client's memory to server's disk, as a short-lived circuit.

CA-NFS uses a reverse auction model. In a reverse auction, the buyer advertises a need for a service and the sellers place bids, like a regular auction. However, the seller who places the lowest bid wins the auction. Accordingly in CA-NFS, when the client is about to issue a request, it compares its local price with the server price. Depending on who offers the lower price the client accelerates, or defers the operation.

We start by describing an auction for a single resource. We then build a pricing function for each resource and assemble these functions into a price for each NFS operation.

3.1 Algorithmic Foundation

For each resource, we define a simple auction in an online setting in which the bids arrive sequentially and unpredictably. In a way, a bid represents the client's willingness to pay for the use of the resource, *i.e.* the client's local price. A bid will be accepted immediately if it is higher than the price of the resource at that time.

Our goal is to find an online algorithm that is competitive to the optimal offline algorithm in any future request sequence. The performance degradation of an online algorithm (competitive ratio) is $r = max\left(B_{offline}/B_{online}\right)$ in which $B_{offline}$ is the benefit from the offline optimal algorithm and B_{online} the benefit from the online algorithm. Awerbuch *et al.* [4] establish the lower bound at $\Omega(\log k)$ in which k is the ratio between the maximum and minimum benefit realized by the online algorithm over all inputs. The lower bound is achieved when reserving $1/\log k$ of the resource doubles the price.

The worst case occurs when the offline algorithm sells the entire resource at the maximum bid P, which was rejected by the online algorithm. For the online algorithm to reject this bid, it must have set the price greater than P, which means it has already sold $1/\log k$ of the resource for at least $P/2$.

$$B_{online} > \frac{P}{2\log k} \text{ and}$$
$$B_{offline} - B_{online} < P$$
$$\implies r < 1 + 2\log k$$

Increasing price exponentially with increased utilization leads to a competitive ratio logarithmic in k.

3.2 A Practical Pricing Function

This model gives us an online strategy that is provably competitive with the optimal offline algorithm in the maximum usage of each resource. It has a weak ($\log$, not constant) competitive ratio, but even this weak ratio is unprecedented in the storage system's literature. The online algorithm knows nothing about the future, assumes no correlation between past and future requests, and is only aware of the current system state.

Based on the theoretical framework, we define the pricing function P_i for an individual resource i in our framework as

$$P_i(u_i) = P_{max}\frac{\{k_i^{u_i} - 1\}}{\{k_i - 1\}}$$

in which the utilization u_i varies between 0 and 1 so that the price varies between 0 and P_{max}.

The parameter k represents the performance degradation experienced by the end user as the resource becomes congested. Thus, appropriate values of k should provide incremental feedback as the resource usage increases.

The heterogeneous resources of distributed file systems complicate parameter selection. Different resources become congested at different levels of utilization, which dictates that parameters need to be set individually. With very large k, the price function stays near zero until the utilization is almost 1. Then the price goes up very quickly. With very small k, the resource becomes expensive at lower utilization, which throttles usage prior to congestion. The network exhibits few negative effects from increased utilization until near its capacity and, thus, calls for a higher setting of k. Similarly, memory works well until it's nearly full at which point it experiences congestion in the form of fragmentation and synchronous stalls from out-of-memory conditions. Disks, on the other hand, require smaller values of k, because each additional I/O interferes with all subsequent (and some previous) I/Os, increasing the service time by increasing queue lengths and potentially moving the head out of position.

CA-NFS users do not need to set the value of k explicitly, as it is precomputed for most existing device types. The pricing mechanism is robust to small hardware variations, *e.g* to different device brands. During various CA-NFS deployments, we experimented extensively with the value of k. (We do not present all these experiments as they are quite tedious.)

We approximate the cumulative cost of all resources by the highest cost (most congested) resource. The highest cost resource corresponds well with the system bottleneck. P_{max} is the same for all server resources and the exponential nature of the pricing functions ensures that resources under load become expensive quickly.

In order to avoid the effects of over-tuning and enforce stability, we set two additional constraints on the cost function. Clients assign an infinitesimally higher value to the maximum price for their resources ($P_{max}+\epsilon$) than do servers. This ensures that when both the client and the server are overloaded, the client sends the operations to the server. In practice, servers deal with overload more gracefully than do clients. Also, the client's prices are always higher than a minimum price P_{min} so that if neither the client nor the server is congested, operations are performed at the server.

3.3 Calculating Resource Utilization

The theoretical model does not make any explicit assumptions about the type of resources managed. As a result, adding new resources to the system is straightforward. We currently monitor the effective usage of five resources, each with its own intricacies:

Server CPU: It is straightforward to establish the utilization of the CPU accurately at any given time through system monitoring.

Client and Server Network: The utilization of networks is also well defined. However, network bandwidth needs to be time-averaged to stabilize the auction. Without averaging, networks fluctuate between utilization 0 when idle and 1 when sending a message. The price would be similarly extreme and erratic. Thus, we monitor the average network bandwidth over a few hundreds of milliseconds.

Server Disk: Measuring disk utilization is difficult because of irregular response times. Although observed throughput seems a natural way to represent utilization, it is not practical because it depends heavily on the workload. A sequential workload experiences higher throughput than a random set of requests. However, disk utilization may be higher in the latter case, because the disk spends head time seeking among the random requests.

We measure disk utilization by sampling the length of the device's dispatch queue at regular, small time intervals. The maximum disk utilization depends on the system configuration. We do not identify the locality among pending operations nor do we use device-specific information. Recently, Fahrad [36] and Zygaria [21] showed the effectiveness of measuring disk utilization by examining the disk head time. We plan to evaluate this approach in future work.

Client and Server Memory: Pricing memory consumption is exceedingly difficult, because memory is a single resource used by many applications for many purposes, caching for reuse, dirty buffered pages, and read ahead. A cache must preserve a useful population of read-cache pages. Deferring writes in CA-NFS could reserve more memory pages to buffer writes, which may in turn reduces cache hit rates, To avoid this, we identify the portion of RAM that is actively used to cache read data and the effectiveness of that cache. We then use pricing to preserve that portion of memory in order to maintain cache hit rates. The price of memory increases if the existing set of pages yields a high cache hit rate or there are a large number of dirty pages that have triggered write-back.

Previous research [6] allows us to effectively track the utility of read cache pages through the use of two ghost caches. We introduce a virtual resource to monitor by using the distribution of read requests among the ghost caches to calculate the projected cache hit rates, and thus, the effective memory utilization. A large fraction of read requests falling in these regions indicates that the client would benefit from more read caching, so deferring writes is not of particular benefit.

Client Read-Ahead Effectiveness: We define a virtual resource that captures the expected efficiency of read-ahead [24, 37]. We build our metric of read-ahead confidence on the adaptive read-ahead logic recently introduced in the Linux kernel [12]. We define confidence as the ratio of accesses to read-ahead pages divided by the total number of pages accessed for a specific file. For high values, the system performs read-ahead more aggressively. For low values, the kernel will be more reluctant to do the next read-ahead.

3.4 CA-NFS Implementation

We have implemented CA-NFS by modifying the existing Linux NFS client and server in the 2.6.18 kernel. Specifically, we added support for the exchange of pricing information and we changed the NFS `write` operation to add support for acceleration and deferral. We have also made modifications to the Linux memory manager to support the classification of the memory accesses and the read-ahead heuristics.

The CA-NFS server advertises cost information to clients, which implement the scheduling logic. We have overridden the `FSSTAT` protocol operation (NFSv3) to include pricing information about server resources. Normally, `FSSTAT` retrieves volatile file system state information, such as the total size of the file system or the amount of free space. Upon a client's `FSSTAT` request, the server encodes the prices of operations based on its monitored resource usage. In our implementation, the server computes the statistics of the resource utilization and updates its local cost information every one second. `FSSTAT` is a lightweight operation that adds practically no overhead to the system resources. Clients do not block waiting for the operation to complete.

Clients send an `FSSTAT` request to the server every ten `READ` or `WRITE` requests or when the time interval

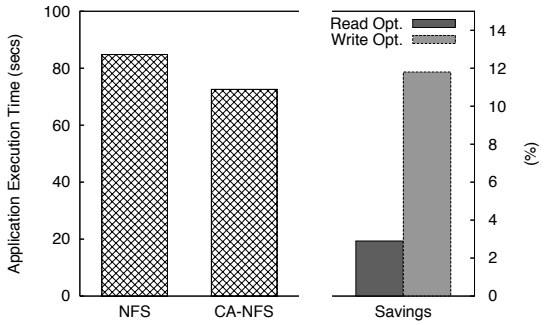


Figure 2: Time to copy a 4GB directory over NFS and CA-NFS and a breakdown of CA-NFS savings

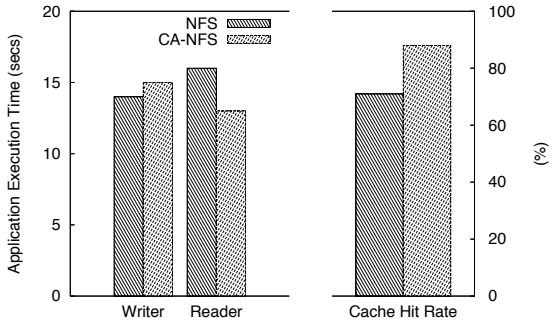


Figure 3: Application execution time and cache hit rates when accelerating writes

from the previous query is more than ten seconds. As part of CA-NFS extensions, we intend to have the server notify active clients via callbacks when its resource usage increases sharply.

4 Evaluation

We run experiments on a cluster of twenty-four machines running at 3.2GHz with 2GB of RAM each. One machine has 4GB of RAM and acts as the server. All nodes are connected via Gigabit Ethernet. To compare CA-NFS with NFSv3, we run a set of micro-benchmarks and application workloads based on the different profiles available in *filebench* [25], Sun's filesystem benchmark, and *IOzone* [18].

4.1 Microbenchmarks

We start our analysis with a simple *filebench* experiment. A single thread of just one client copies a large directory of 4GB over CA-NFS and NFS. This workload creates a hierarchical directory tree, then measures the rate at which files can be copied from the source tree to the new tree. The sizes of the files in the directory vary from 1KB to 200MB. Even in such a simple configuration, CA-NFS provides 15% improvement in performance, measured by completion time (Figure 2).

Regular NFS clients fail to use their local memory to good effect even though it is not congested. NFS clients read data from the server and start buffering write pages until they reach the statically defined limit of dirty pages. Then, the flushing daemon forces the pages to be written to the server. This requires the server to harden data to disk. The resulting write traffic delays disk read requests. In contrast, CA-NFS clients determine that the server disk is heavily utilized through the exchange of pricing information. CA-NFS clients use a much larger portion of their RAM to buffer dirty pages, avoiding the large, asynchronous writes to the server that interfere with reads. The effects of read-ahead optimizations are less dramatic, but still important. Read-aheads are issued aggressively in the beginning, because there is free memory space and they yield a high hit rate for this mostly sequential workload. As the server memory resources become more congested with dirty pages, client-initiated read-aheads are performed more prudently. Figure 2 also breaks out the portion of the improvement attributed to write (12%) and read-ahead optimizations (3%).

4.1.1 Operation Scheduling

Accelerating writes: The next experiment combines two *IOzone* workloads to show how CA-NFS preserves cache hit rates by valuing client memory highly. We consider a client application that writes a 2GB file sequentially. On the same client, another application performs re-reads, i.e. reads that will be server cache hits if the system does not evict the pages.

Figure 3 shows the execution times of the two applications for NFS and CA-NFS. CA-NFS improves read performance by 21% when compared with NFS. The NFS client evicts memory pages used for read caching in order to buffer writes. This reduces the cache hit rate and application-perceived read performance as a consequence. NFS clients replace approximately 15% of the pages used for caching and realize a cache hit rate of only 70%. In contrast, CA-NFS accelerates writes by flushing them immediately, anticipating the importance of the cache contents. CA-NFS clients maintain a cache hit rate of 90%. The client prefers to accelerate all asynchronous writes, because its read cache is producing a high hit rate, thus its price for asynchronous writes is high. The server price for asynchronous writes is low, because none of its resources is congested.

Deferring writes: We now demonstrate how CA-NFS uses write-buffering at the client to avoid I/O interference at the server. One client issues random reads that are serviced by the server's disk. Another client writes a 1GB file to the server. The NFS client sends the write requests

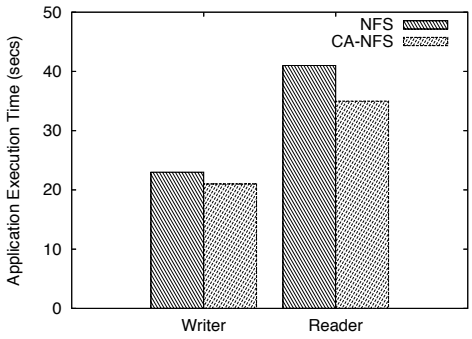


Figure 4: Execution time for two clients reading and writing to a file when deferring writes

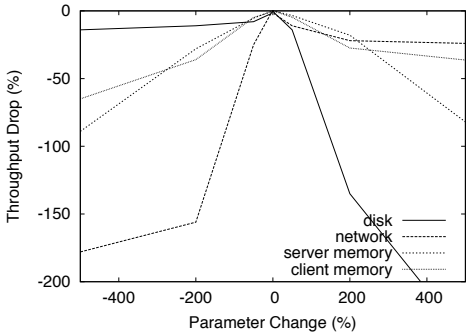


Figure 6: CA-NFS throughput sensitivity to the correct parameterization of k

to the server, which flushes them to stable storage. These disk writes increase the service time of disk reads, because they interfere at the disk with read requests coming from the first client. Through pricing, CA-NFS identifies congestion at the disk, which causes the writing client to buffer dirty data and reduces the amount of write data delivered to the server. Figure 4 shows that CA-NFS improves read performance by 18%. In this case, write performance is also improved by 6%.

4.1.2 On the Pricing Metric

We characterize how the pricing function captures system dynamics by comparing resource utilization and resource price side-by-side. We show that pricing reflects congestion on heterogeneous resources, i.e. on networks, for memory, and on the disk. Prices create a single view of system load in a resource independent manner.

For the network resource, we run a network intensive workload with four clients reading a 1GB file from server's memory at a rate of 50MB/sec each over a GbE network. Three clients suffice to saturate the network bandwidth. We start each client at 10 second time intervals in order to provide incremental load to the system. Figure 5(a) plots the server-perceived throughput and the average throughput at the clients. Figure 5(b) shows the server's system price, governed by the network, at the same time scale. As the system load increases, each client gets a smaller share of the bandwidth and average client throughput drops. Over time, the network price increases to near it maximum value (1.0, the value of P_{max} in all experiments). The increase in price causes the clients to back off, preventing overload, and the server throughput remains stable under heavy load.

We run a similar experiment for memory-bound workloads and memory price. A client issues reads by increasing the number of requests for already accessed data. By recycling the client's memory, we force all re-read requests to be serviced out of the server cache only. From the server's perspective, as the hit ratio increases cache data become more important, so it increases the price for memory operations (Figures 5(c) and 5(d)).

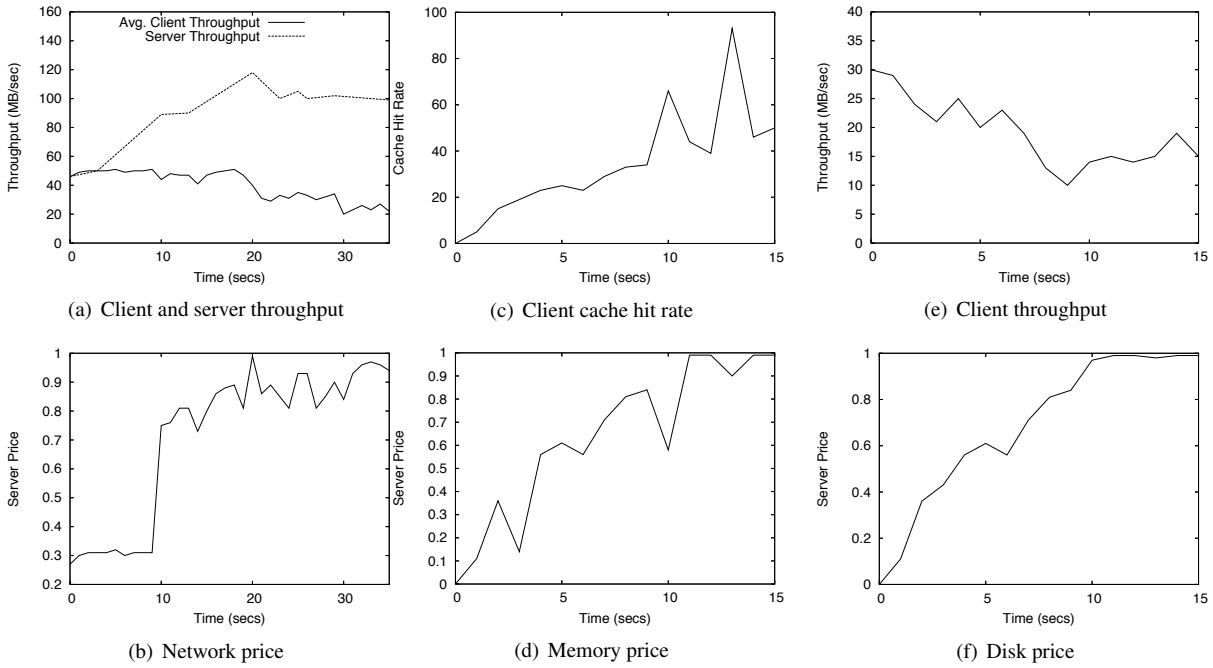
For disk-bound workloads, we run a client process that issues random read requests over increasingly larger spans of the disk. As we increase the span, client throughput drops from increased disk head utilization that leads to more requests in the disk dispatch queue (Figure 5(e)). In response to the increase in the number of pending requests, the system increases the price for the disk resource (Figure 5(f)).

In the next experiment, we show how the selection of parameter k affects system performance. We run a read-write *IOzone* workload on two clients accessing a 2GB file on the server. Through measurements, we have established a value of k for each device type, which yields the best throughput for this experiment. We alter the value of k for the client and server memory, the network and the disk resources, and we examine the drop in system throughput.

Figure 6 shows that small perturbations of k do not affect CA-NFS performance. However, if the value of k differs significantly from its optimal (as calculated) setting, performance degradation is notable. Low values of k lead to underutilization of the system resources, while high values make the system less adaptive, as prices increase very rapidly. Figure 6 also shows that the disk and the network are more sensitive to correct parameterization. This is because, these resources exhibit very high (disk) or very low interference (network) between past and future requests. As already mentioned, CA-NFS users do not have to set the value of k explicitly. In the next set of experiments, we show that the CA-NFS parameter selection is robust to different workload types.

4.2 Application Benchmarks

Microbenchmark experiments demonstrate the operation of CA-NFS by isolating the benefits of individual optimizations. To better understand how CA-NFS effects applications, we turn our attention to macrobenchmarks.



(a) Client and server throughput (c) Client cache hit rate (e) Client throughput

(b) Network price (d) Memory price (f) Disk price

Figure 5: Examining the pricing mechanism for three different resources (network (a,b), memory (c,d), and disk (e,f))

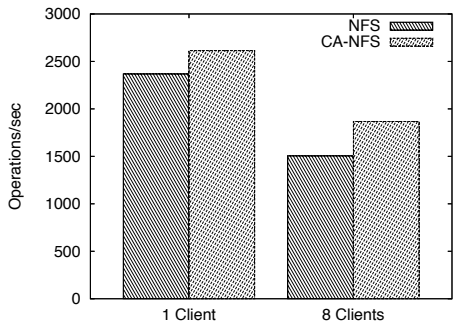


Figure 7: Average number of ops/sec per client for the file-server benchmark

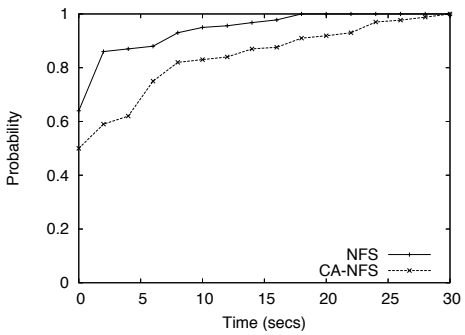


Figure 8: CDF of the time that the system schedules write-backs for NFS and CA-NFS

First, we evaluate CA-NFS by running the `fileserver` synthetic workload provided by *filebench*, on eight clients. This workload is modeled after SPECsfs [39], an industry standard test suite that is based on data collected by SFS committee members from thousands of real NFS servers operating at customer sites. The test performs a sequence of creates, deletes, appends, reads, writes and attribute operations on the file system. We randomly set the number of user threads, the number of files written and the average file size to numbers between 100-200, 1000-5000 and 100-5120KB respectively. This workload contains a large number of asynchronous operations.

Figure 7 shows that CA-NFS outperforms NFS by more than 10% in the single client setup and by more than 20% in the eight-client setup. Figure 8 shows the cumulative distribution function of the time that elapses between a write operation submitted by the application and the relevant pages marked for commit by the file system. CA-NFS schedules asynchronous write operations very differently from NFS. NFS clients are forced to commit many pages almost immediately as they become dirty, in order to prevent the system from running out of memory to buffer dirty pages. No page stays dirty for more than 12 seconds after the write is issued. CA-NFS schedules the write-back operations more evenly across the 30-second time frame that defines the reliability window for asynchronous writes in most current operating systems. As a result, traffic in CA-NFS is less bursty, a significant factor that improves performance.

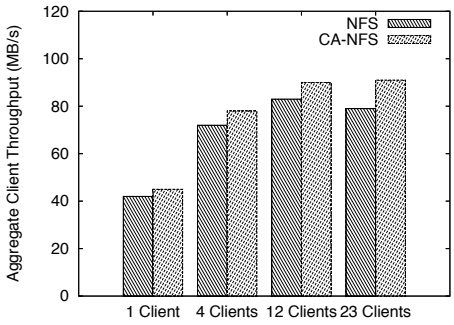


Figure 9: Aggregate client throughput for the oltp benchmark a function of the number of clients

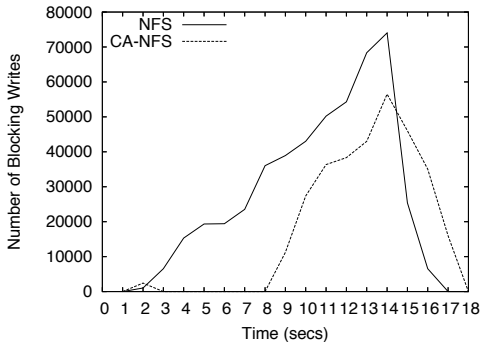


Figure 10: Number of asynchronous client writes blocked

	NFS	CA-NFS
ops/sec		
	2443	2503
ms/op		
open file	34.3	33.7
read file	5.3	5.1
close file	4.0	4.2
append log	7.1	7.2

Table 1: NFS and CA-NFS under the webserver workload

The next benchmark examines CA-NFS characteristics under an OLTP workload that performs transactions into a filesystem using an I/O model from Oracle 9i. This workload tests for the performance of small random reads and writes in conjunction with moderate (128KB) synchronous writes. Operations represent read and write OLTP transactions and writes to the log file respectively. On each client, we launch 200 reader processes, 10 processes for asynchronous writing, and a log writer. We run the experiments four times, modifying the number of active clients.

For the `oltp` workload, CA-NFS is more scalable than NFS. Although this workload exhibits some cache locality on the server, the main bottleneck in this experiment is the server's disk, which is overwhelmed by the number of incoming requests. Figure 9 plots the aggregate client throughput for different client populations. For a small number of clients (one to four), CA-NFS provides a rather small performance advantage. As the number of clients increases, the relative throughput of CA-NFS increases when compared with NFS. In the case of NFS, the aggregate throughput for the 23-client setup is less than in the 12-client setup. This is because the number of incoming requests overwhelms the server resulting in a throughput crash.

In our last experiment, we examine the performance of CA-NFS under a workload that contains mostly synchronous read operations. The file server exports its filesystem to a number of web servers. The `webserver` workload from the *filebench* suite consists of a mix of open/read/close of multiple files in a directory tree, plus a file append (to simulate the web log) in which 16KB is appended to the weblog for every 10 reads.

CA-NFS performs slightly better (about 3%) than NFS thanks to the read-ahead optimizations. Write optimizations are not a factor in this benchmark, because of the small number of write operations. Table 4.2 shows that for all operations the latency for both NFS and CA-NFS is almost identical.

Macrobenchmark experiments show that CA-NFS significantly outperforms NFS under workloads with a significant number of asynchronous operations, such as the `fileserver` benchmark. For workloads that are read-dominated (`webserver`) or contain small, asynchronous requests (`oltp`), CA-NFS performs comparably to NFS, showing that our modifications are lightweight.

4.3 High-Speed Hazards

To further evaluate our framework, we perform measurements over a 10-Gbps Infiniband network. As opposed to the previous set of experiments, in this setup, the network bandwidth outstrips disk transfer rates. We consider the two clients writing file data sequentially to the server for fifteen seconds over NFS and CA-NFS. During the write burst, both clients write data at the maximum rate, close to 200MB/sec.

This experiment shows that running out of memory turns asynchronous file system operations into synchronous that block all progress (Figure 10). Regular NFS experiences synchronous waits for asynchronous writes starting at 2 seconds. When the number of dirty pages on the NFS clients reaches the flushing point, clients start writing data, which overwhelms the disk system and memory available to buffer writes at the server fills. All subsequent writes block awaiting completion.

CA-NFS detects congestion on the server memory and I/O system through pricing and buffers writes in local memory. This makes the effective write-buffering space 8GB, 2GB on each client and 4GB on the server, rather than the 4GB of server memory that NFS uses. CA-NFS does not experience synchronous waits until 8 seconds and blocks fewer writes overall. This results in higher overall throughput as well. CA-NFS writes 4.8GB worth of data whereas regular NFS writes only 3.9GB, an improvement of 23%.

This scenario shows how the emergence of high-speed networking makes holistic storage management critical. For storage systems, we are on the verge of a new era. Infiniband and 10Gbps Ethernet deliver data at such rapid rates that storage systems that receive and process these data cannot keep up. This gap between network bandwidth and disk throughput creates a memory crisis for storage servers. Many clients writing data in parallel will create a data stream that a server cannot transfer to disk. Flow control in the transport protocol will be irrelevant, because the system is not network bound. The server buffers data pending I/O completion and the buffered data accumulate until memory is full. This results in a cascading throughput crash over the entire system [11].

5 Future Directions

Although our focus is on the scheduling of asynchronous operations, pricing synchronous operations wisely can enable the system to manage nonstandard I/O processes. Distributed file systems often have lower-priority I/O tasks, such as data mining, indexing, backup, etc. Capping the willingness to pay for synchronous operations causes these low-priority tasks to halt automatically when resources become congested. Clients can also encode application priorities and differentiate between critical and noncritical tasks by charging different processes different prices.

The proposed framework does not address the issue of fairness over time. Operation costs are proportional to the current state of the system but independent of the client that put the system into the state. For example, one client could fill the server cache with dirty data, pushing up prices for all others.

Finally, more complex resource management goals can be realized by adding constraints to the auction model. For example, resource reservations can be accomplished by differentially pricing the same resource among clients. The goal is to insulate one client from the consumption by non-reserving clients. To do so, we need to limit the spending of non-reserving clients and increase resource prices prior to exhaustion, creating an artificial shortage. Also, proportional sharing arises when clients are given salaries, i.e. a rate of consumption or fixed amount of spending over some time interval. This concept extends the ideas of flow control beyond networks to cover all resources in the system. Pricing certain resources and making all other resources free, allows sharing to be targeted to specific resources only.

6 Related Work

Economic Models: Using economic models for resource management is not a novel approach [10]. Auction-based systems have been applied in a broad range of distributed systems including clusters [9], computational grids [26], parallel computers [40], and Internet computing systems [27]. These systems are intended for coarse-grained resource allocations.

Network Flow Control: Flow control schemes offer to each client a proportional share of the network and, thus, guarantee to a large extent fairness [31]. Many different approaches exist in the literature, including TCP-like window based protocols [14, 19], feedback schemes [13], and optimization based methods [15].

The congestion pricing techniques upon which we build have been used by Amir *et al.* [2] to manage a single network resource. Kelly [23] was the first to describe pricing for flow and congestion control. However, our approach and Amir's are algorithmic, whereas Kelly relies on economic theory.

Memory Management: Li *et al* [28] acknowledge the asynchronous nature of writes and their dependence on the client's state. They propose a scheme where the storage clients inform the storage servers about the types of writes that they perform by passing write hints. These write hints can then be used by the server to manage the second-tier cache.

Carson and Setia [7] showed that for many workloads, periodic updates from a write-back cache perform worse than write-through caching. They suggest two alternate disciplines: (1) giving reads non-preemptive priority and (2) interval periodic writes in which each write gets its own fixed period in the cache. Mogul [32] implements an approximate interval periodic write-back policy that staggers writes in time using a small (one second) timer. Golding *et al* [16] delay write-back until the system reaches an idle period. This reduces the delays seen by reads by postponing competing writes until idle periods, possibly with the help of nonvolatile memory, in order to ensure consistency.

Storage controllers with nonvolatile memory employ adaptive destaging policies that vary the rate of writing [1, 43] or the destage threshold [33, 43], based on memory occupancy and filling and draining rates. In these systems, cached writes are persistent, so they want to delay destaging data as long as possible.

Patterson *et al* [35] in TIP made cache residency and prefetching decisions over the network following a cost benefit analysis. Their work was based on models that value memory pages for different type of data, such as prefetched, buffered, or cached. Nelson *et al* [34] in Sprite mentioned a weight used to trade off how to partition memory between pages for the file cache and for virtual memory. Nelson's principle was not applied to a distributed context. These approaches use heuristic methods and do not look at the relative load across all clients.

Storage Scheduling and QoS: Storage quality of service (QoS) attempts to optimize the system resources individually [17, 29] or conjunctively [22]. Fairness in the QoS context is generalized to incorporate weights used to introduce deliberate bias, depending for example on different service-level agreements (SLAs) [45].

In general, quality of service (QoS) approaches are not well-suited for multi-resource optimization. CA-NFS complements QoS methods [8, 22, 30, 42] that employ I/O throttling in order to limit resource congestion and avoid throughput crashes. We do not offer the performance guarantees to applications on which one might build SLAs [29]. Instead, we follow a best-effort approach to improve application-perceived performance by minimizing latency and maximizing throughput for synchronous file system operations.

Provisioning: Provisioning systems use a single metric, utility or cost in dollars, to unify heterogeneous resources when deciding the initial configuration of a system under a fixed utility budget. Recently, Strunk et al. [41] provide a framework for provisioning based on detailed system models and genetic algorithms to explore the configuration space. This extends the previous work on provisioning of Anderson*et al.* [3].

While the unification of resources using utility is superficially similar to pricing, provisioning solves a very different problem. Provisioning determines how to achieve the best availability, throughput, or IOPS under a fixed budget as a static offline configuration problem. CA-NFS examines dynamic pricing of operations under changing workloads in static configurations.

7 Conclusions

We have shown the importance of using holistic performance management for the adaptive scheduling of lower-priority distributed file system requests based on system congestion in order to reduce their interference with foreground, synchronous requests. We also show the virtue of adaptation based on application-perceived performance, rather than server-centric metrics.

CA-NFS introduces a new dimension in resource management by implicitly managing and coordinating the usage of the file system resources among all clients. It unifies fairness and priorities in a single framework that assures that realizing optimization goals will benefit file system users, not the file system servers.

Acknowledgements

We would like to thank the NetApp Kilo-Client support team, and especially Alan Belanger, for providing the environment in which we tested CA-NFS. Also, we thank our shepherd, Narasimha Reddy, for all of his help. This work was supported in part by NSF awards IIS-0456027 and CCF-0238305.

References

[1] M. Alonso and V. Santonja. A new destage algorithm for disk cache: DOME. In *EUROMICRO Conference*, 1999.

[2] Y. Amir, B. Awerbuch, C. Danilov, and J. Stanton. A cost-benefit flow control for reliable multicast and unicast in overlay networks. In *IEEE/ACM Transactions on Networking*, 2005.

[3] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 2005.

[4] B. Awerbuch, Y. Azar, S. A. Plotkin, and O. Waarts. Competitive routing of virtual circuits with unknown duration. In *Symposium on Discrete Algorithms*, 1994.

[5] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *ACM Symposium on Operating Systems Principles*, 1991.

[6] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey. AWOL: An adaptive write optimizations in layer. In *Conference on File and Storage Technologies*, 2008.

[7] S. D. Carson and S. Setia. Analysis of the periodic update write policy for disk cache. *IEEE Transactions on Software Engineering*, 18(1), 1992.

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

[9] B. N. Chun and D. E. Culler. User-centric performance analysis of market-based cluster batch schedulers. In *CCGRID '02: IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[10] S. Clearwater. Market-based control: A paradigm for distributed resource allocation. *World Scientific*, 1996.

[11] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Symposium on Operating Systems Design and Implementation*, 1996.

[12] W. Fengguang. Adaptive read-ahead in the Linux kernel. `http://lwn.net/Articles/155097/`.

[13] S. Floyd. TCP and explicit congestion notification. *ACM Computer Communication Review*, 24(5):10–23, 1994.

[14] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[15] R. Gibbens and F. Kelly. Resource pricing and the evolution of congestion control. *Automatica*, 1999.

[16] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness Is Not Sloth. In *USENIX Annual Technical Conference*, 1995.

[17] P. Goyal, D. Jadav, D. S. Modha, and R. Tewari. CacheCOW: QoS for storage system caches. In *International Workshop on Quality of Service (IWQoS 03)*, 2003.

[18] The IOzone Benchmark. http://www.iozone.com.

[19] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM*, 1988.

[20] M. B. Jones, D. Rou, and M. Rou. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Symposium of Operating Systems and Principles*, 1997.

[21] T. Kaldewey, T. Wong, R. Golding, A. Povzner, S. A. Brandt, and C. Maltzahn. Virtualizing disk performance. In *Real-Time and Embedded Technology and Applications Symp*, 2008.

[22] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance isolation and differentiation for storage systems, 2004.

[23] F. Kelly, A. Maulloo, and D. Tan. Rate control in communication networks: Shadow prices, proportional fairness and stability. In *Journal of the Operational Research Society*, volume 49, 1998.

[24] A. Ki and A. E. Knowles. Adaptive data prefetching using cache information. In *International Conference on Supercomputing*, 1997.

[25] E. Kustarz, S. Shepler, and A.Wilson. The new and improved filebench file system benchmarking framework. *Conference on File and Storage Technologies*, 2008.

[26] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent Grid Syst.*, 2005.

[27] L. Levy, L. Blumrosen, and N. Nisan. On line markets for distributed object services: the majic system. In *USITS'01: USENIX Symposium on Internet Technologies and Systems*, 2001.

[28] X. Li, A. Aboulnaga, K. Salem, A. Sachendina, and S. Gao. Second-tier cache management using write hints. In *Conference on File and Storage Technologies*, 2005.

[29] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for qos guarantees and its application to differentiated caching services. In *International Workshop on Quality of Service*, 2002.

[30] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual storage devices with performance guarantees. In *Conference on File and Storage Technologies*, 2003.

[31] L. Massoulie and J. Roberts. Bandwidth sharing: Objectives and algorithms. In *INFOCOM*, 1999.

[32] J. Mogul. A better update policy. In *USENIX Summer Technical Conference*, 1994.

[33] Y. J. Nam and C. Park. An adaptive high-low water mark destage algorithm for cached RAID5. In *Pacific Rim Internataional Symposium on Dependable Computing*, 2002.

[34] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the sprite network file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[35] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *ACM Symposium on Operating Systems Principles*, 1995.

[36] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. *SIGOPS Oper. Syst. Rev.*, 42(4):13–25, 2008.

[37] D. Revel, D. McNamee, D. Steere, and J. Walpole. Adaptive prefetching for device independent file I/O. Technical Report CSE-97-005, Oregon Graduate Institute School of Science and Engineering, 1997.

[38] A. Riska, E. Riedel, and S.Iren. Adaptive disk scheduling for overload management. In *International Conference on the Quantitative Evaluation of Systems*, 2004.

[39] The SPEC SFS97R1 (3.0) Benchmark. Available at http://www.spec.org/sfs97r1.

[40] I. Stoica, H. Abdel-Wahab, and A. Pothen. A Microeconomic Scheduler for Parallel Computers. In *Workshop on Job Scheduling Strategies for Parallel Processing*, 1995.

[41] J. .D. Strunk, E. Thereseka, C. Faloutsos, and G. R. Ganger. Using utility to provision storage systems. In *Conference on File and Storage Technologies*, 2008.

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

[43] A. Varma and Q. Jacobsen. Destage algorithms for disk arrays with nonvolatile caches. *IEEE Transactions on Computers*, 47(2), 1995.

[44] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned scalable internet services. In *Symposium of Operating Systems Principles*, 2001.

[45] Z. Zimmermann and U. Killat. Resource marking and fair rate allocation. In *International Conference on Communications*, 2002.