FreeLoader: Scavenging Desktop Storage Resources for Scientific Data

Sudharshan S. Vazhkudai* Xiaosong Ma†* Vincent W. Freeh† Jonathan W. Strickland†
Nandan Tammineedi† Stephen L. Scott*

Abstract

High-end computing is suffering a data deluge *from experiments, simulations, and apparatus that creates overwhelming application dataset sizes. End-user workstations—despite more processing power than ever before—are ill-equipped to cope with such data demands due to insufficient secondary storage space and I/O rates. Meanwhile, a large portion of desktop storage is unused. We propose the FreeLoader framework, which aggregates unused desktop storage space and I/O bandwidth into a shared cache/scratch space, for hosting large, immutable datasets and exploiting data access locality. This paper presents the FreeLoader architecture, component design, and performance results based on our proof-of-concept prototype. Our experiments show that FreeLoader is an appealing low-cost solution to storing massive datasets, by delivering higher data access rates than traditional storage facilities: namely, local or remote shared file systems, storage systems, and Internet data repositories. In particular, we present novel data striping techniques that allow FreeLoader to efficiently aggregate a workstation's network communication bandwidth and local I/O bandwidth. In addition, the performance impact on the native workload of donor machines is small and can be effectively controlled.*

Keywords: *Distributed storage, storage scavenging, storage cache, serverless storage system, scientific data management, parallel I/O, striped storage*

1 Introduction

There has been a phenomenal increase in computational power, however, the increase in application data size is even greater [19]. This growing gap is highlighted in desktop computing environments, which remain indispensable in scientists' everyday research activity, especially for interactive tasks such as simulation-results visualization and experiment data analysis. While numerous applications are within the computation capability of a moderately powerful workstation, often the application cannot run due to the shortage of storage space.

This imbalance between computation and storage leaves scientists with three unattractive choices. First is direct processing on remote data. This is slow (due to remote-end processing power and wide area networking) and not always possible. Second is using a local high-performance machine equipped with more storage resources, as a computation backend for desktop data processing. This is inconvenient, under-utilizes the expensive machine, and requires significant extra programming and maintenance work. The last option is installing a local storage system (e.g., a SAN or NAS). This is an expensive, storage-only acquisition (although disks can be acquired at roughly $1000/TB, 4TB SAN storage currently costs in the upwards of $40,000), which is often not affordable or justifiable in academic and government research environments.

Meanwhile, a large amount of disk space remains idle on personal computers. Studies show that on average, half of the disk space on desktop workstations is idle, and the fraction of idle space increases as the disks become larger [2, 14]. In addition, most workstations are online for the vast majority of the time [7]. A desirable and low-cost alternative then, is to **harness the collective storage potential of individual workstations** much as we harness idle CPU cycles [28]. Besides aggregating storage capacity, this brings performance benefits as well: as networking trends suggest that a fast LAN connection can stream data faster than local disk I/O, a workstation can get higher data throughput by effectively performing parallel I/O on multiple workstations where its data is distributed.

We envision a distributed storage framework, *FreeLoader* (Figure 1), that provides abundant, high-performance site-local storage for scientific datasets with very little additional expense, by aggregating idle desktop storage resources. With FreeLoader, workstation owners—within a local area network—donate

*Computer Science and Mathematics Division, Oak Ridge National Laboratory {vazhkudaiss,scottsl}@ornl.gov
†Department of Computer Science, North Carolina State University {xma,vwfreeh,jwstric2,ntammin}@ncsu.edu

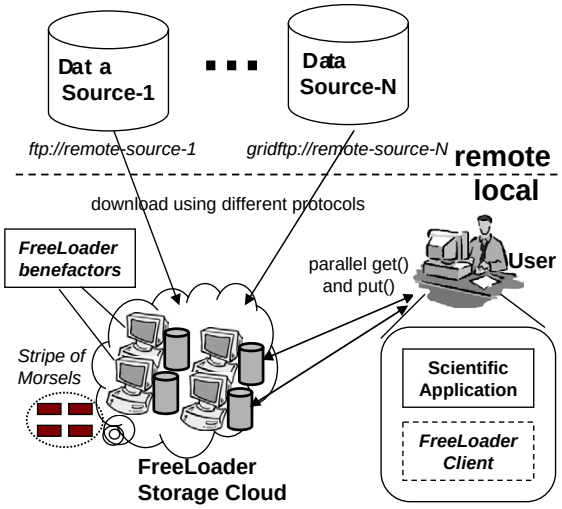


Figure 1. Envisioned FreeLoader Environment

some disk space, and FreeLoader stripes datasets onto multiple such workstations (called *benefactors*) to enhance data access rates. Imagine a group of scientists in an organization—working on a problem of mutual interest—who regularly run their simulations on a remote supercomputer to generate dozens of gigabytes of snapshot-data per timestep. They often upload these terabytes of result-data onto local machines to study using visualization tools numerous times for a period of weeks.

On the other hand, using FreeLoader these researchers can pool the idle disk space on their workstations into a transparent, shared *cache* and *scratch* space. This enables each researcher in the group to process the raw datasets as if they reside on a high-performance shared file system, allowing easy collaboration and obviating expensive downloading/migration operations. As interest fades on this batch of datasets, they will get replaced by new datasets that are currently "hot."

Although there exists other work on desktop storage aggregation [2, 5], FreeLoader is novel in two aspects. First, rather than providing a general-purpose distributed file system, FreeLoader is a lightweight software cache/scratch space that recognizes unique characteristics of interactive data-intensive computing.

- Datasets are usually write-once-read-many. Further, they are usually shared as people within the same organization, *e.g.*, a research group or academic department, often have shared interest on certain datasets [34].
- Often scientists have the primary copy of a dataset safely stored in a remote repository, typically at archiving or file systems attached to a parallel computer, or at data collections on the web [30, 38, 40].
- A certain dataset is of interest for a limited period, *e.g.*, a few days or weeks. It may be frequently re-visited during this period, often by multiple coworkers-workers in the collaboration [25]. However, beyond this processing duration, users normally choose not to retain copies of the downloaded datasets locally.
- Workstations that are used to perform scientific data analysis or visualization tend to have more performance and resources.

As a result, FreeLoader is designed to handle transient uses of bulk scientific data, rather than to be a general-purpose distributed file system. It aggregates idle storage to host datasets that are larger than workstations' typical local disk space, and employs an asymmetric striping technique to fully take advantage of local space and I/O bandwidth at workstations that process data from FreeLoader space.

Second, because FreeLoader aggregates workstation storage where users also conduct their day-to-day activities, it is vital to control the performance impact on donated nodes. This paper reports initial experimentation measuring the performance impact of disk scavenging, which suggests that FreeLoader induces reasonable and containable impact on a variety of native workloads. A prior publication addresses controlling the impact on the benefactors native workload [39].

The rest of the paper presents the design, implementation, and evaluation of our FreeLoader prototype.

2 Related Work

Tens of networked and distributed file systems exist as shared storage (*e.g.*, NFS [33]), LOCUS [35], CODA [24], etc.). These systems either use centralized servers (as in NFS) or a few distributed replicated file servers (as in CODA). Several serverless file systems are designed to achieve higher availability and scalability (*e.g.*, Farsite [2] and Kosha [5]). However, all the above systems serve as file systems and target general-purpose file system usage patterns. Additionally, GFS [18] is a distributed storage system designed for data-intensive tasks, but is proprietary, uses dedicated disks, and is specialized for Web searches. In contrast to these existing systems, FreeLoader is an open-source, lightweight, highly decentralized storage *cache* built on scavenged disk spaces. It aims to host large replicated datasets for data-intensive science, where concerns for file/directory management and concurrency control are much less significant.

Parallel file systems (*e.g.*, GPFS [37], Lustre [9], and PVFS [6]) target large datasets, provide sustained

high I/O throughput, and are tightly integrated with supercomputers. These systems are widely used by FreeLoader's target customers: scientists engaged in high performance computing. FreeLoader applies parallel file system techniques, such as file striping and parallel I/O, in desktop storage settings, and complements these high-end systems. By replicating datasets at scientists' local sites, FreeLoader improves data availability, facilitates local data sharing, and reduces the I/O and network workload at those remote file systems. Meanwhile, by serving as a data cache, FreeLoader achieves high space utilization and avoids wasting or fragmenting (due to space quotas) its capacity. Also, while data striping has been widely adopted in the above systems and certain distributed systems (*e.g.*, Zebra [22]), it is usually done in a uniform or symmetric way with the relatively homogeneous settings of these systems. FreeLoader explores overlapping network data transfer and local I/O with a novel *asymmetric striping* technique.

To some extent, FreeLoader can be viewed as cooperative caching [13, 17, 36] extended to another layer: it pools secondary storage in a LAN environment to reduce access misses that require wide-area data transfer from remote sources. However, a cooperative cache is part of the storage hierarchy of every node, whereas FreeLoader space is donated voluntarily and can be aggregated to *enable* the local desktop processing of large datasets.

Finally, multiple large scale P2P [12] systems exist (*e.g.*, Gnutella [29], Kazaa [31], Freenet [8] and BitTorrent [10]). PAST [15] and OceanStore [16] facilitate wide-area distributed data storage by providing persistence and reliability. Also akin to our approach is Squirrel [26], a decentralized P2P web cache, that exploits locality in web data object references by sharing desktop browser caches. There are two major differences between P2P systems and FreeLoader. First, P2P systems are usually designed for WAN settings and emphasize scalable resource and replica discovery, routing protocol, and consistency. In contrast, FreeLoader focuses on aggregating space and bandwidth in a corporate LAN setting. It adopts a certain degree of centralized control in data placement and replication, for better data access performance. Second, P2P storage systems are designed for *content sharing*, while FreeLoader has an additional goal of *space aggregation*. Although P2P systems can be deployed in LAN environments, individual workstations that have such a system installed still manage their own storage spaces. This is also true for P2P web caching. In contrast, FreeLoader has total control over scavenged space and can therefore aggregate space effectively to host *large* and *hot* datasets: a workstation may host a dataset that its owner never downloads or uses, or lose a dataset without its owner explicitly deleting it. Moreover, the access pattern of scientific datasets [34], differs significantly from that of P2P file sharing systems [20], often designed toward multimedia data consumption.

Table 1 compares FreeLoader with some of the related existing systems. In summary, compared to existing systems, FreeLoader possesses a novel combination of several techniques: it deploys space scavenging to aggregate storage resources in non-dedicated commodity workstations in a LAN environment and performs aggressive and asymmetric data striping for better data access performance. Instead of being a general-purpose file system, it works as scratch/cache space to exploit data access locality, as well as to enhance space utilization in data-intensive scientific computing.

3 Architecture

FreeLoader aggregates donated storage into a single storage system. The basic architecture consists of two layers. The *management* layer maintains the metadata and performs high-level operations, such as replication and cache replacement. The *storage* layer consists of *benefactor* nodes that donate space along with I/O and network bandwidth. Data storage and retrieval are initiated by *client* nodes that interact with managers and benefactors to access data. A client node may or may not be a benefactor itself.

FreeLoader is a storage system, not a file system. It stores large, immutable datasets by fragmenting them into smaller, equal-sized chunks called *morsels*, which are scattered among the benefactors. This allows easy load balancing and striping between benefactors for better overall throughput. The morsel size presents a tradeoff between flexibility and overhead. Our preliminary experiments with 1MB morsels have proven practical for FreeLoader managing hundreds of GBs to TBs of space.

3.1 Management Layer

The management layer maintains metadata (such as dataset names) and performs lookup services to map a client-requested dataset to morsels on benefactors. This layer does not touch any data in the dataset. Because the amount of metadata is significantly smaller than real data, the management layer can run on one or a handful of dedicated machines—this fact is exploited in a similar way by Google FS, at a system scale of thousands of nodes [18]. Clients communicate with a manager node to obtain morsel mapping, then directly contact the benefactors for morsel transfer.

Besides morsel location lookups, the management layer stores client-specific metadata for added function-

Systems	General-purpose FS	Cache	Striping	Desktop Storage Scavenging	Wide-area
FreeLoader	No	Yes	Yes	Yes	No
Parallel file systems					
GPFS [37], Lustre [9], PVFS [6]	Yes	No	Yes	No	No
Distributed file systems					
Frangipani [41]+Petal [27], Zebra [22]	Yes	No	Yes	No	No
NFS [33], AFS [23], Coda [24]	Yes	No	No	No	Yes
Google FS [18]	Yes	No	No	No	No
FARSITE [2]	Yes	No	No	No	No
IBP+exNode [3]	No	No	Yes	No	Yes
P2P Storage					
Kosha [5]	Yes	No	No	Yes	No
BitTorrent [10]	No	No	Yes	No	Yes
Gnutella [29], Kazaa [31], Freenet [8]	No	No	No	Yes	Yes
Squirrel [26]	No	Yes	No	No	No
Cooperative caching					
xFS [13], GMS [17], hints [36]	No	Yes	No	No	No

Table 1. Comparison with related file and storage systems. The column "General-purpose FS" indicates whether a system is designed to present general file system interfaces and functionalities. The column "Cache" indicates whether the system is intended to be used as a cache space, instead of whether the system uses caching (many of them do emphasize caching for performance improvement). "Striping" denotes the use of data placement optimizations. "Desktop storage scavenging" indicates the use of space contributions either in part or whole. "Wide-area" column denotes use as wide-area storage.

ally, such as per-morsel fingerprint checksums for increased dataset integrity. Such services, including encryption/decryption, are optional client-side filters that have little storage overhead at the management layer. Moreover, the computational costs are paid by clients (not benefactors) who elect to use them.

For aggregating I/O bandwidth, FreeLoader adopts software striping [22] by distributing morsels to multiple benefactors. In addition to aggregating disk and network transfer bandwidth, striping has one unique benefit in FreeLoader: it lowers performance impact on benefactors by spreading out data requests.

When distributing data to remote benefactors, FreeLoader adopts a simple round robin striping approach, where *stripe width* is the number of benefactors that a dataset is striped onto, and *stripe size* is the number of contiguous morsels assigned to a benefactor in each round of striping. For each individual dataset, determining these two parameters is a complex decision based on a set of factors: network connectivity of the client, free space and bandwidth of available benefactors, reliability and native workload on these benefactors, etc. Section 5 shows the impact of these striping parameters on FreeLoader's data access rates.

Moreover, recognizing the fact that the user who imports or creates a dataset is most likely the one who visualizes or analyze it often, we designed an asymmetric striping approach that assigns more data to this benefactor workstation, to optimize its future accesses to the dataset by overlapping remote data retrieval and local I/O. Section 4.1 discusses asymmetric striping in more detail.

Several other features, not presented at length in this paper, are currently under development. In short, reliability and availability is addressed by recovery and data replication mechanisms. Manager recovery is based on periodical metadata checkpointing and fail-over techniques. Benefactor failures, including *sudden death* (due to crashes) and *planned leave* (due to space withdrawal), are handled using a combination of cache replacement and replication mechanisms. To this end, FreeLoader collects and uses data access patterns and benefactor performance capabilities extensively.

3.2 Storage Layer

The storage layer, which runs on benefactor nodes, manages all the morsels in the system. The primary function of this layer is servicing *get* and *put* morsel requests. Because FreeLoader stores read-only datasets and accesses to scientific datasets have temporal locality [34], get requests will dominate traffic.

A benefactor node is an ordinary user machine that has donated certain idle disk space and has installed the FreeLoader benefactor component as a daemon process, which services *get/put* morsel requests. The benefactors will also perform several aggregate or meta operations at the direction of the manager. For example, in the case of data relocation, the manager gives the source benefactor a list of morsels to move out and their destination benefactors. The source benefactor initiates the transfers and reports back to the manager.

FreeLoader makes no assumption on the availability of individual benefactor nodes. Soft-state registration is performed by having each benefactor regularly sends heartbeat or "I'm alive" messages to the manager(s).

Another important task of the storage layer is performance impact control on benefactors' native workloads.

Aside from servicing requests, the impact of the daemon is negligible. When it comes to servicing morsels, the performance impact depends on the bandwidth or request frequency, as well as on the native workload's resource usage pattern.

Typical impact control strategy for resource stealing systems is *all-or-nothing*: a scavenger has all the resources at its disposal if there are no native tasks, and no resources otherwise [1, 28]. Such a strategy is not only over conservative [21, 32], but also infeasible for FreeLoader by incurring intolerably long data access latencies whenever benefactor owner activities are detected. Therefore FreeLoader is designed to have the benefactor daemon's data serving co-exist with native workloads, with active control of the performance impact. FreeLoader contains impact to a pre-specified threshold by performance impact benchmarking, real-time monitoring of the native workload's resource consumption, and throttling the benefactor daemon's execution. Interested readers are referred to our paper [39].

In this paper, we develop a high-level approach such as increasing the stripe width to control benefactor impact. Stripe width increase naturally performs impact control by reducing the per-benefactor data request size and complements aforementioned local impact control. Section 5 demonstrates our empirical performance impact study and control through striping.

4 Prototype Implementation

Our FreeLoader proof-of-concept prototype (Figure 2) implements major functionalities described in Section 3 and verifies the following rationales.

- Harnessing workstation storage delivers aggregate data retrieval rates at least comparable to those currently possible using existing local or remote storage systems.
- Software striping delivers both high aggregate data access throughput and scalability with regard to stripe width in a LAN environment. In particular, with asymmetric striping, a client can combine its network data transfer with local disk I/O.
- Data serving activities exert a tolerable impact on workstation's native workload and this impact is controllable.
- The overhead of the FreeLoader framework, not considering bulk data transfer, is acceptably low and reduces as stripe width increases.

4.1 Manager

This prototype deploys a management layer consisting of a single manager. However, it is designed for multiple managers. For simplicity, this section

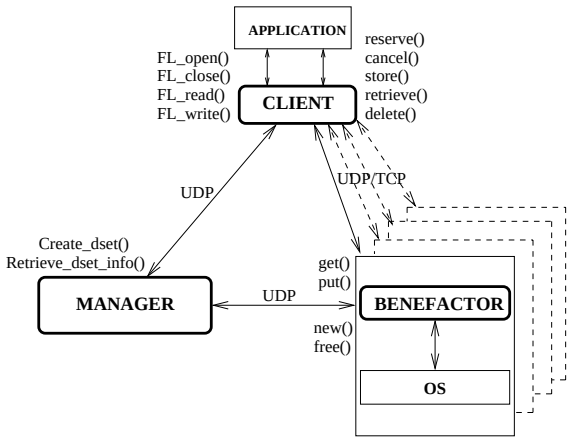


Figure 2. Modules and interfaces in prototype

discusses a single manager because that is what was tested. The manager provides a set of services for storage resource scavenging and data accesses: global free space management, space reservation/cancellation, data striping and metadata serving in the process of dataset store/retrieve operations. Below, we discuss these services at length.

The manager keeps track of space donations—both available and occupied—at each benefactor, in number of morsels. A client needs to make a space reservation with the manager before storing a dataset in FreeLoader. This provides space guarantees before expensive data imports, and acts as a serialization point for concurrency control between multiple client requests.

During a store or retrieve operation, the client obtains morsel distribution from the manager. This information is organized as an array of {*benefactor ID*, *morsel ID*} pairs, specifying for each morsel-sized block in the dataset, the benefactor storing this block and the local morsel ID assigned by that benefactor. This format allows for flexibility in striping data and future data relocation in case of benefactor failures. Such metadata is cached in the manager's memory and further backed up in its secondary storage.

Upon a store, the manager performs file striping across benefactors by choosing a stripe width and size, as well as the subset of benefactors on which the dataset will be placed. In this prototype, we have the client specify these two parameters for a dataset to be stored, allowing easy experimentation on combinations of stripe parameters. An intelligent striping algorithm should manage space efficiently while also factoring in performance capabilities of benefactors. We have implemented two data striping strategies: *round-robin* to benefactors excluding the client that stores the dataset, and *asymmetric* that stripes to both benefactors and this client (called

host client of the dataset in question).

Round-Robin striping: FreeLoader's data placement optimization problem, which we call *Stripe Fit*, can be formalized as follows. A sequence of n datasets $D = d_1, d_2, ..., d_n$, where d_i is of size s_i and requests a stripe width w_i, arrive to be stored at a set of m benefactors $B = \{b_1, b_2, \ldots, b_m\}$, where b_i comes with an initial free space size f_i. The problem then is to stripe as long a prefix as possible of D to B. This is the point where FreeLoader has to perform cache replacement. We have shown that a known NP-hard problem, *Minimum Bin Packing*, can be reduced to the off-line version of this Stripe Fit problem.[1]

FreeLoader has to make on-the-fly decisions as datasets arrive. For this purpose, we implemented a greedy algorithm, where the manager sorts the benefactors by their current free space sizes. Each dataset, d_i, is striped to the top w_i benefactors on the sorted list. If a dataset is too large to be accommodated at any w_i benefactors, the above sorting and striping are repeated for the overflow part, until the entire dataset is stored. This way, we automatically perform load balancing between benefactors, ensure that each dataset is accessed simultaneously from only w_i benefactors, and minimize the total number of benefactors involved with each dataset for better data availability.

Asymmetric Striping: The above striping technique does not exploit the capabilities and access patterns of the host client that imported the dataset. In reality, the owner of the host client workstation is likely to access it first and more frequently. In addition, workstations that are used to performing bulk scientific data processing (visualization and/or analysis) tend to have higher configurations in memory size, bus bandwidth, network interface, and disk space. To better utilize the resources of the host client of each dataset stored in FreeLoader, we have developed an asymmetric data placement strategy that, in addition to striping data on a width of w_i benefactors (as in round-robin), also treats the local downloading client as a benefactor by placing part of the dataset locally.

This approach serves two purposes. First, it aggregates throughput from the width of benefactors as well as overlap that with local disk I/O. The upshot is a potential throughput gain that is substantially larger than either storing the dataset locally or on a width of benefactors in isolation. Second, with a predisposition towards host clients while placing datasets, overall network traffic can be reduced due to the aforementioned access locality.

Thematic to this approach, however, is the *local:remote* data ratio. This ratio determines how many morsels will be stored locally and remotely (on the w_i remote benefactors) respectively in each stripe cycle. We have confirmed that the optimal ratio to obtain good data retrieval performance roughly corresponds to the local I/O rate and aggregate network transfer rate from the remote benefactors. However, the local:remote data ratio is subject to capacity constraints as well. We approach this in our implementation with a two-phase technique. First, we determine the optimal ratio, check whether the host client has enough donated space to accommodate such data locally, and adjust the ratio if there is not sufficient local space. For data striped to the other benefactors, we use the round-robin striping process as described above. With a given local:remote ratio, the local morsels are distributed uniformly with remote morsels, so that local I/O requests are scattered between benefactor accesses.

Another aspect to consider with asymmetric striping is the cost it incurs on other clients: while a prejudice in data placement works to the advantage of the host client, it may create load imbalance when the dataset is accessed by another client. Our results (see Section 5.2) show that this cost is not significant. Moreover, asymmetric striping is implemented as a user option, which is set when the dataset is first stored in FreeLoader depending on anticipated access pattern. For example, a scientist may turn asymmetric striping on when importing simulation results that she expects to study alone, and turn it off when importing a biological sequence database against which all group members routinely perform searches.

4.2 Benefactor

The benefactor is a user-level daemon consisting of four major components: a communication library, a scavenger device, a morsel service layer and a monitoring layer. Figure 2 highlights a few APIs for each of the components.

A reliable communication library, built atop UDP, services requests, and transfers metadata and other status information between nodes. UDP, with its low overhead, is better positioned to serve these short and transient messages. Message types and their associated handlers are registered with the library. Upon receiving a message, an associated handler is invoked.

The scavenger device component manages metadata that maps datasets and their morsels onto local files. It keeps track of local free space using a bitmap. Morsels from the same dataset are stored in order in a single file, which reduces seek time considering the prevailing sequential accessing pattern in scientific data processing.

The morsel service layer transfers raw data to and from the benefactor, through the *get* and *put* interfaces,

[1] For detailed proof, see http://www.csm.ornl.gov/vazhkuda/-Morsels/proof.html.

as shown in Figure 2. It also performs support operations such as local space allocation (*new*) and release (*free*). We choose to transfer morsels using TCP, because bulk transfers benefit from the reliability and congestion/flow control mechanisms that TCP has to offer. In one dataset store/retrieve operation, the TCP connections between the client and appropriate benefactors are cached and re-used for subsequent morsel transfers. Thus, only the first morsel transferred incurs a slow-start phase in TCP.

The monitor layer, currently only supported under Linux, is used in performance impact control. Using the */proc* file system, it observes changes in usage of the CPU, memory, network, and disk. Such real-time information can help the benefactor throttle its data service rate and "yield" to native workloads, as suggested by results in Section 5.

4.3 Client

The major goal of the client component is to efficiently parallelize data transfers across benefactors. In this paper, we focus our discussions on dataset retrieval performance because datasets stored in FreeLoader are write-once-read-many, and the storing speed is often bound by retrieval rates from remote data sources (such as using FTP).

Data retrieved from FreeLoader is either stored on the client's disks or stream processed by a program. Both local processing tasks benefit from assembling morsels retrieved into long, sequential segments. We use an efficient buffering strategy to overlap data transfers from multiple benefactors, overlap network data transfer with local processing, and perform data assembling. The client requests morsels from benefactors, and maintains a fixed buffer pool of size at least $w_i \times (s_i + 1)$ morsels. This way, a generalized double buffering scheme allows network and I/O activities to proceed in parallel. We implemented a pair of nonblocking morsel retrieval interfaces, `getMorsel` and `waitAny`, to enable the client to multiplex efficiently between benefactors and maintain w_i outstanding morsel requests. The morsel buffer pool is shared between these w_i TCP connections through a cyclic queue, allowing benefactors to proceed at different speeds. The client outputs filled morsel buffers for local processing between sending morsel requests and performing `waitAny`. It promotes sequential processing by waiting for filled morsels in the buffer pool to form contiguous blocks (*i.e.*, without "holes").

We implemented a client wrapper library for standard I/O function calls (*e.g.*, open, close, read, write) in C, as a prelude to a kernel file system module. This library creates familiar interfaces for client programs to access datasets stored in FreeLoader. The open call's semantic is interesting in that it sets the stage for subsequent reads/writes by querying the manager for a dataset's morsel distribution information. The read and write calls are translated into FreeLoader morsel transfer operations, with additional processing such as data trimming and concatenation. The close call performs cleanup. Section 5 demonstrates a widely-used application using these interfaces in processing a biological sequence database stored in FreeLoader.

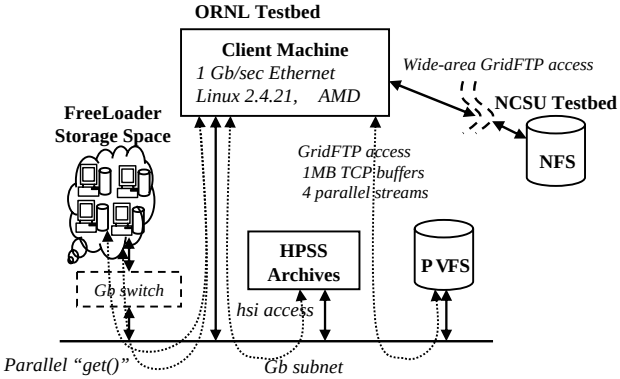


Figure 3. FreeLoader testbed

5 Results

This section presents results of our prototype implementation in three parts: client-side perceived FreeLoader data access performance, running an example application which streams data from FreeLoader space, and the performance impact that a benefactor daemon places on a donated machine.

5.1 Testbed Configuration

Our testbed (Figure 3) depicts a scientist's HPC research environment with a powerful, well-connected local client machine, with access to parallel/archival file systems and high-speed data movement tools. We installed the FreeLoader storage cache in this setting as shown in Figure 3 to study its use by a researcher in his HPC setting. Our testbed spreads across both Oak Ridge National Laboratory (ORNL) and North Carolina State University (NCSU), and comprises of the following: (1) FreeLoader cloud at ORNL: Aggregate storage of 400GB, 15 benefactors (donating 7-60GBs each) and one manager. Benefactor config: Dual Pentium III, Linux 2.4.20-8 kernel and 100 Mb/sec Ethernet. (2) The PVFS parallel file system [6] on the ORNL TeraGrid Linux cluster outside ORNL firewall with several terabytes of storage (accessed through GridFTP [4]).(3) The HPSS archival storage system [11] at ORNL with 365PB of tape storage and several hundreds of gigabytes of high-speed disk caches (accessed through hierarchical storage interface (hsi) client). (4) The NFS

FreeLoader	15 Benefactors, 1 Manager and 1 client
(Data set size)	256MB,512MB,1GB,2GB,4GB,16GB,32GB,64GB
(Morsel size)	1MB
(Stripe size)	1MB, 2MB, 4MB, 8MB, 16MB, 32MB
(Stripe Width)	1, 2, 4, 6, 8, 9
PVFS	GridFTP, GSI authent., 1MB TCP buffer
NFS	GridFTP, GSI authent., 1MB TCP buffer
HPSS	hsi with DCE authentication
(Hot)	Data maintained in disk caches of HPSS
(Cold)	Fetch from tape: data purged from caches before fetch
NCBI	wget from `http://www.ncbi.nlm.nih.gov`

Table 2. Throughput test setup

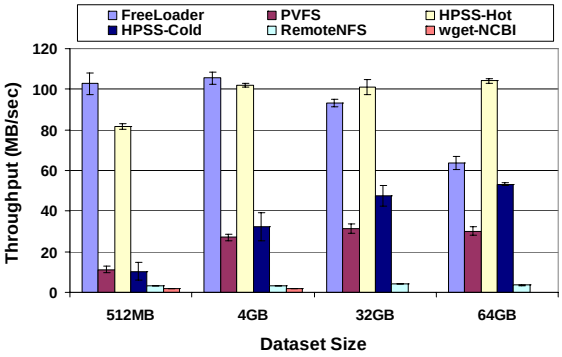


Figure 4. "Best of class" comparison of data retrieval throughput, with 95% confidence ranges. FreeLoader throughput is an asymmetric striping on eight benefactors. wget-ncbi datasets were unavailable for larger sizes. In the rest of experiments, FreeLoader's performance variance is similar to that depicted in this figure and error bars are omitted.

shared file system at the NCSU HPC center's blade cluster (accessed through GridFTP). (5) A client machine at ORNL: Dual AMD Opteron, Linux 2.4.21 and GigE. The client is at most five hops away from any of the benefactor nodes in the FreeLoader cloud, the PVFS and the HPSS. This machine runs the FreeLoader client component. (6) A gigabit subnet at ORNL to which the FreeLoader storage cloud, PVFS, HPSS and the client machine are connected, which is in turn connected to an OC-12 link for external connectivity.

5.2 FreeLoader Performance

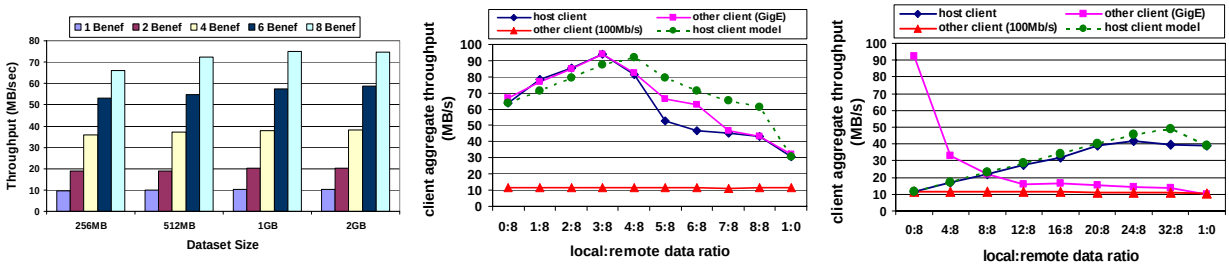
First, we analyze the performance of the FreeLoader storage cloud and compare it against alternative data sources (NFS, PVFS, HPSS and Internet scientific repositories [30]) frequently used by scientists. We conducted several transfers experiments over a week (see summary in Table 2) and report average results.

Figure 4 compares the "best of class" performance using FreeLoader (asymmetric striping: client + 8 benefactors) and other data sources. For all dataset sizes, FreeLoader performs better than GridFTP-based PVFS and hsi-based HPSS "cold" accesses. We observed up to a threefold throughput advantage with FreeLoader for larger datasets and a much higher difference for smaller datasets (2GB or less). This is because, both PVFS and HPSS are highly optimized for bulk data transfers. FreeLoader's performance was comparable to HPSS "hot" accesses. HPSS hot access simulates a near optimal throughput obtained due to transfers entirely from high-speed disk caches, on a gigabit subnet by two GigE connected entities (the client and HPSS). However, the majority of HPC users do not have access to on-site HPSS, and HPSS's disk cache is shared by a much larger group of users than a typical FreeLoader instance will have. FreeLoader matches such a GigE-transfer by aggregating throughput from low-end individual benefactors. For the largest dataset size (64GB) in our experiment, FreeLoader throughput decrease is because we were unable to sustain a stripe width of 8 for the entire dataset (w_i=8 up to 48GB, after which w_i drops to 3). This was due to storage capacity ceiling on the benefactor contributions. Scientific datasets, however, are usually large collections of smaller files and can be accommodated and scalably serviced by the FreeLoader framework with reasonable contributions from several donor machines. Not surprisingly, when compared to remote data sources, such as the NFS at NCSU and the NCBI website, FreeLoader has a over an order of magnitude throughput advantage.

These results show that, in addition to benefits such as space aggregation and data sharing, FreeLoader has significant performance advantages. By utilizing collective workstation storage in a networked environment, which is likely to be used by a much smaller group of users compared to file/archival systems attached to large clusters or web servers, FreeLoader can become an attractive alternative to scientists.

Next, we study the effect of striping parameters. By increasing the stripe width (Figure 5(a) with stripe size of 1MB morsel), we see an almost linear throughput improvement, due to better utilization of the residual bandwidth available at the client and extra I/O bandwidth. By varying the stripe size, however, we see no noticeable improvements in our experiments (throughput results are therefore omitted due to space concern).

In the following discussion on asymmetric striping, we refer to the client uploading the dataset into FreeLoader space as the "host" client and all other clients accessing the datasets as "other" clients. Figure 5(b) and 5(c) show the effect of asymmetric striping in retrieving a 16GB dataset striped to the host client and 8 remote benefactors, with a GigE- and 100Mb/s-connection machine as the host client respectively. The x axis shows varying local:remote data ratio ($size_l$: $size_r$). Obviously, the ratio 0:8 stands for round-robin



(a) Stripe width variation, with round-robin striping on benefactors

(b) Asymmetric striping with GigE host client for 16GB dataset

(c) Asymmetric striping with 100Mb/s host client for 16GB dataset

Figure 5. FreeLoader striping results

striping on benefactors only, with throughput $thpt_r$. Similarly, the ratio 1:0 stands for local I/O only, with throughput $thpt_l$. To evaluate how well local-area network data accesses can be overlapped with local I/O, we plot in the figures using dotted line a simple model for the host client's overall throughput: $thpt_{overall} = (size_l + size_r)/Max(\frac{size_l}{thpt_l}, \frac{size_r}{thpt_r})$. In addition, we show data retrieval throughput measured from the host client and two other clients that do not store any parts of the dataset, again with GigE and 100Mb/s interface respectively.

In both settings, the host client's dataset access rate follows the trend of the idealized model, achieving nice overlap between remote data access and local I/O. The measured access rate does reach the peak value slightly earlier than the model, most likely due to better file system prefetching effect when the local I/O requests are slowed down by the host client's handling remote accesses. The GigE and 100Mb/s host clients need very different optimal local:remote ratio, which can be derived approximately at store time using our throughput model, or with a diagnostic test similar to these experiments when a workstation joins FreeLoader. In particular, in both cases the peak throughput with asymmetric striping is significantly higher than the local I/O rate, motivating the use of FreeLoader even when users do have enough local disk space, for higher access rates.

When it comes to other clients, accessing datasets optimally-placed for host clients, the two settings show different impact of asymmetric striping. For the client with 100Mb/s interface, its bottleneck is the network connection and its access rate remains flat despite the biased data distribution on benefactors (flat line on both Figures 5(b) and 5(c)). The GigE client, however, benefits from the GigE host client serving more data and shows a similar rate curve as the host client (Figure 5(b)). On the other hand, it experiences severe throughput drop as more data gets stored on the 100Mb/s host client (Figure 5(c)). The above behavior is not surprising since all the other benefactors in this case have 100Mb/s connection. In summary, the positive or negative impact asymmetric striping incurs on a "3rd party" client depends on the client's and the benefactors' configuration, but is predictable. At the store time of a dataset, these factors could be evaluated in conjunction with the expected access pattern for an optimized striping plan.

5.3 Sample Application

Besides client APIs for storing/retrieving entire datasets, we have also implemented a small subset of file system interfaces to access datasets in FreeLoader space. This allows us to stream-process data cached by FreeLoader. We evaluate this FreeLoader service by running a data-intensive application: *formatdb* from the NCBI BLAST toolkit, which preprocesses a raw biological sequence database to create a set of sequence and index files. These output files are used in subsequent sequence alignment searches. Since the input raw database is normally larger than all the *formatdb* output files combined, and can be formatted in different ways, it is the ideal type of data that users may want to cache/share in the FreeLoader space.

	Local	NFS	Stripe width		
			1	2	4
Throughput (MB/s)	1.71	1.75	1.71	1.82	1.84

Table 3. Overall throughput, in MB/s, for *formatdb* to process a 1GB sequence database

Table 3 shows *formatdb* execution time with three input data sources: local file-system, network file-system (NFS), and FreeLoader. For FreeLoader, we tested stripe widths of 1, 2, and 4. This example is a proof-of-concept; it shows that an application can transparently use FreeLoader and receive some benefit. The overall throughput demanded by *formatdb* is small. However, because it comes in bursts, the performance increases as we stripe across benefactors. With one benefactor, FreeLoader performs about the same as the local file-system and 2% slower than NFS. With 4 benefactors, FreeLoader is 5% faster than NFS. We do not see much

improvement beyond 4 benefactors due to network and client saturation.

5.4 Performance Impact on Benefactor Nodes

We have shown that FreeLoader can be an attractive storage choice from clients' view point. What about from space donors' view point? This section evaluates the performance impact on benefactors' native workloads, by measuring the slowdown factor of three typical types of activities: computation, network, and disk, caused by morsel-serving. In each test, a benchmarking client requests morsels at various rates, from 0 morsels per second to the maximum sustainable bandwidth (which varies depending on user workload). Tests were conducted on a benefactor node that represents an "average" desktop machine, not too powerful or too weak, with a 2.8GHz Pentium 4, a SATA disk, 512MB of memory, and a 100Mb/s network connection. Results show averages and 95% confidence intervals from multiple runs.

For the computation impact test, we performed two tests: (1) the EP application from the NAS benchmark,[2] and (2) a Linux kernel compile. The latter is not completely CPU-bound, but represents typical computation-intensive user tasks. Figure 6(a) shows their normalized execution time as the benefactor servicing load increases. In general, the impact is low. Even when serving morsels at full speed, EP is slowed down by 14% and compilation by 21%, compared to without FreeLoader benefactor running. Currently, we are unable to explain the anomalous behavior of "compile" at 1MB/s.

Our network activity test simulates a user downloading several different sized web pages from different servers, located from 3 to 19 hops away from the benefactor. Consequently, the latency to fetch each page varies depending on the size and location of the server. Each page was requested using `wget` hundreds of times back to back, to make it (hopefully) cached by the web server but not by the web client on the benefactor. Figure 6(b) shows very small to moderate impact on these downloads, depending on the page size and location. With the exception of one data point, the latency increase is at most 37%, and for loads of 6MB/s or less, 23%. The exception occurs for a large, remote file (19 hops) and only near the maximum sustainable benefactor load. The benefactor's data serving is not impacted much, because it stresses the uploading rather than downloading network bandwidth.

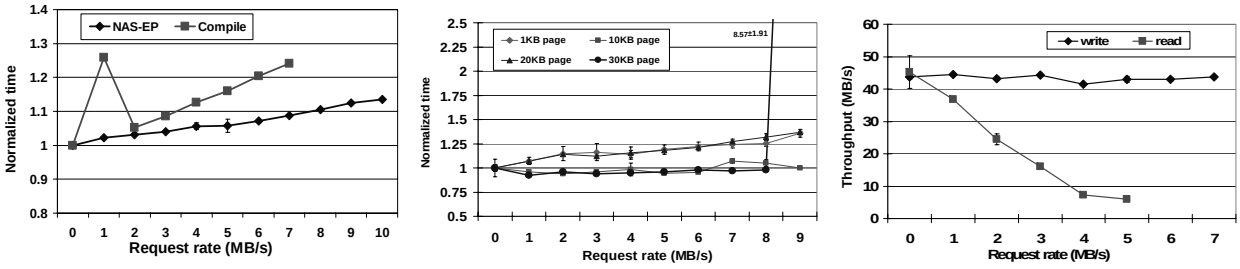
Our disk activity test simulates a user reading/writing a 1GB file. We flush the memory when necessary to remove the file-system cache effect. While desktop users do not typically read/write such large files, it stresses I/O and delivers a worst-case contention at the disk when the native workload is reading un-cached data. Figure 6(c) shows a steady decrease in the user disk read throughput, until 20% of its original throughput when the morsel request rate is 4MB/s. Meanwhile, the maximum sustainable throughput at the FreeLoader benefactor side is less than 5MB/s. On the other hand, the user write throughput stays constant under any load, with a maximum FreeLoader benefactor bandwidth of slightly more than 9MB/s. This asymmetry is because the OS delays and combines write requests. Compared to blocking read operations, writes are more resilient to concurrent disk activities.

In summary, Figure 6 shows that FreeLoader's performance impact on typical native workloads is fairly low. More importantly, it reveals that in most cases, its performance impact grows smoothly with the morsel request rate, allowing FreeLoader to actively perform impact control, as demonstrated below.

Recognizing that I/O contention brings the highest performance impact, and that users are mostly affected in interactive tasks, we built an I/O intensive composite workload to simulate interactive user activities with intervals. A static idle period of 1-3 seconds was set between executing the following operations: 1) Writing 25 MB of randomly-generated data to files in a specific directory. This simulates unzipping a downloaded file into a local directory. 2) Browsing arbitrary system directories in search of a file. 3) Compressing the written data from the first part of the simulation with bzip into a file and transferring this file across the network to a data repository. 4) Browsing a few more local directories. 5) Finally removing all data files from the beginning of the simulation. The following operations were executed in a tight loop a few times, taking a total of 115 seconds without any other concurrent user workload on our chosen benefactor.

We ran the above composite workload on one of the benefactors concurrently with the client's retrieval of a 2GB dataset. This will impact both the composite native workload and the client's perceived aggregate data access throughput. Figure 7 depicts such an impact from both sides with varying stripe width (asymmetric striping is not used in this test). At the benefactor, it shows the percentage of slowdown compared with the time to completion of the composite native workload when executed alone (115 seconds). At the client, it shows the percentage of throughput loss compared with the client aggregate data retrieval throughput *using the corresponding stripe width without the composite workload on any of the benefactors*. As stripe width increases, the benefactor side impact goes down steadily. From stripe width 1 to 2, the data retrieval time is longer than the composite workload, and the decrease in benefactor

[2] http://www.nas.nasa.gov/Software/NPB/



(a) Computation (b) Network (c) Disk

Figure 6. Benefactor impact results

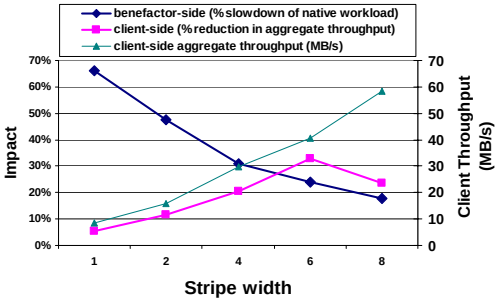


Figure 7. High-level benefactor impact control by increasing the striping width. Primary y-axis plots benefactor slow down and client throughput loss ratio. Secondary y-axis plots actual client throughput.

impact comes from reduced data request rate from the client. Beyond that point, another factor comes into play due to increased stripe width: the total dataset retrieval time keeps decreasing, so that the endurance of performance impact on the native workload is shortened. This factor also contributes to the growth of client-side impact, as larger portions of the retrieval is affected by the slowed-down benefactor. This effect is overcome when the stripe width increases to over 6. Meanwhile, the absolute client aggregate throughput grows steadily as stripe width increases. This shows that striping serves both purposes as benefactor bandwidth aggregation and impact control. Again, more aggressive impact control can be performed at the benefactors [39].

6 Conclusions

This paper demonstrates the design of the FreeLoader storage aggregation framework. Our experiment results show that FreeLoader is an attractive storage alternative for scientists to cache and share their datasets locally, with good performance and low, controllable performance impact on storage resource donors. Our major contributions are listed below.

- We proposed and built a novel framework for aggregating idle, existing commodity storage resources, that complements high-end storage systems in caching large scientific datasets.
- Validated distributed software striping in FreeLoader, and developed novel approaches to perform asymmetric data placement in order to optimize client achievable throughput.
- Measured the performance impact of storage scavenging on space donors' native workloads.

References

[1] Seti@home: The search for extraterrestrial intelligence. http://setiathome.ssl.berkeley.edu/, 2003.

[2] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J.Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[3] M. Beck, T. Moore, and J. Plank. An end-to-end approach to globally scalable network storage. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2002.

[4] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, 1999.

[5] A. Butt, T. Johnson, Y. Zheng, and Y. Hu. Kosha: A peer-to-peer enhancement for the network file system. In *Proceedings of Supercomputing*, 2004.

[6] P. Carns, W. Ligon III, R. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[7] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5), 2003.

[8] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage

and retrieval system. *Lecture Notes in Computer Science*, 2000.

[9] Cluster File Systems, Inc. Lustre: A scalable, high-performance file system. http://www.lustre.org/docs/-whitepaper.pdf, 2002.

[10] B. Cohen. Incentives Build Robustness in BitTorrent. 2003.

[11] R.A. Coyne and R.W. Watson. The parallel i/o architecture of the high-performance storage system (hpss). In *Proceedings of the IEEE MSS Symposium*, 1995.

[12] J. Crowcroft and I. Pratt. Peer to Peer: peering into the future. In *NETWORKING Tutorials*, 2002.

[13] Michael Dahlin, Randolph Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Operating Systems Design and Implementation*, pages 267–280, 1994.

[14] J. Douceur and W. Bolosky. A large-scale study of file-system contents. In *Proceedings of SIGMETRICS*, 1999.

[15] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, 2001.

[16] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[17] Michael J. Feeley, William E.Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, , and Chandramohan A. Thekkath. Implementing global memory management in a workstation cluster. In *In Proc. of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[18] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of the 19th Symposium on Operating Systems Principles*, 2003.

[19] J. Gray and A. S. Szalay. Scientific Data Federation. In I. Foster and C. Kesselman, editors, *The Grid 2: Blueprint for a New Computing Infrastructure*, pages 95–108, 2003.

[20] P. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the 19th Symposium on Operating Systems Principles*, 2003.

[21] A. Gupta, B. Lin, and P. Dinda. Measuring and understanding user comfort with resource borrowing. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, 2004.

[22] J. Hartman and J. Ousterhout. The Zebra striped network file system. In *Proceedings of the 14th Symposium on Operating Systems Principles*, 1993.

[23] J. H. Howard. An overview of the andrew file system, 1998.

[24] http://www.coda.cs.cmu.edu. *CODA File System*, 1987.

[25] A. Iamnitchi, M. Ripeanu, and I. Foster. Small-world file-sharing communities. In *Infocom*, 2004.

[26] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.

[27] E. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[28] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, 1988.

[29] E. Markatos. Tracing a large-scale peer to peer system: An hour in the life of gnutella. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2002.

[30] National center for biotechnology information. http://-www.ncbi.nlm.nih.gov/, 2005.

[31] SHARMAN NETWORKS. The kazaa media desktop. http://www.kazaa.com.

[32] R. Novaes, P. Roisenberg, R. Scheer, C. Northfleet, J. Jornada, and W. Cirne. Non-dedicated distributed environment: A solution for safe and continuous exploitation of idle cycles. In *Proceedings of the Workshop on Adaptive Grid Middleware*, 2003.

[33] B. Nowicki. *NFS: Network File System Protocol Specification*. Network Working Group RFC1094, 1989.

[34] E. J. Otoo, D. Rotem, and A. Romosan. Optimal file-bundle caching algorithms for data-grids. In *Proceedings of Supercomputing*, 2004.

[35] G. Popek and B. J. Walker. *The LOCUS Distributed System Architecture*. MIT Press, 1985.

[36] Prasenjit Sarkar and John Hartman. Efficient cooperative caching using hints. In *Proceeding of the 2nd ACM Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, 1996.

[37] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies*, 2002.

[38] Sloan digital sky survey. http://www.sdss.org, 2005.

[39] J. Strickland, V. Freeh, X. Ma, and S. Vazhkudai. Governor: Autonomic throttling for aggressive idle resource scavenging. Proceedings of the 2nd IEEE Int'l Conference on Autonomic Computing, 2005.

[40] A. Szalay and J. Gray. The world-wide telescope. *Science*, 293(14):2037–2040, 2001.

[41] C. Thekkath, T. Mann, and E. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.