Constructing Collaborative Desktop Storage Caches for Large Scientific Datasets

SUDHARSHAN S. VAZHKUDAI
Oak Ridge National Laboratory
XIAOSONG MA
North Carolina State University
VINCENT W. FREEH
North Carolina State University
JONATHAN W. STRICKLAND
North Carolina State University
NANDAN TAMMINEEDI
North Carolina State University
TYLER SIMON
Oak Ridge National Laboratory
and
STEPHEN L. SCOTT
Oak Ridge National Laboratory

Author's address: S. Vazhkudai, T. Simon, S. Scott, Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831. X. Ma, V. Freeh, J. Strickland, and N. Tammineedi, Department of Computer Science, North Carolina State University, Raleigh, NC, 27695-7534.

High-end computing is suffering a *data deluge* from experiments, simulations, and apparatus that creates overwhelming application dataset sizes. This has led to the proliferation of high-end mass storage systems, storage area clusters, and data centers. These storage facilities offer a large range of choices in terms of capacity and access rate, as well as strong data availability and consistency support. However, for most end-users, the "last mile" in their analysis pipeline often requires data processing and visualization at local computers, typically local desktop workstations. 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. Its architecture comprises contributing benefactor nodes, steered by a management layer, providing services such as data integrity, high performance, load balancing, and impact control. 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. Further, we show that security features such as data encryptions and integrity checks can be easily added as filters for interested clients. Finally, we demonstrate how legacy applications can use the FreeLoader API to store and retrieve datasets.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.4 [**Performance of Systems**]:

General Terms: Storage Networking, Storage Resource Management

Additional Key Words and Phrases: Distributed storage, storage scavenging, storage cache, serverless storage system, scientific data management, parallel I/O, striped storage

1. INTRODUCTION

Increasingly, scientific discoveries are driven by analyses of massive data, produced by instruments or computer simulations [Gray and Szalay 2003; Gray et al. 2005; Avery and Foster 2001]. This has led to the proliferation of high-end mass storage systems, storage area clusters, and data centers as storage fabric elements for Grids, and Internet scientific data collections. These storage systems offer a wide range of choices in terms of capacity, access rate, access control, support for high-throughput parallel I/O, optimization for wide-area bulk transfers, and data reliability.

However, for most end-users of scientific data, certain stages of their tasks often **require computing, data processing, or visualization at *local* computers**, typically personal desktop workstations. A local workstation is and will remain an indispensable part of end-to-end scientific workflow environments, for several reasons. First, it provides users with interfaces to view and navigate through data, such as images, timing and profiling data, databases, and documents. Second, users have more control over hardware and software on their personal computers compared to on shared high-end systems (such as a parallel computer), which allows much greater flexibility and interactivity in their tasks. Third, personal computers provide convenience in connecting users' computing/visualization tasks with other tools used daily in their work and collaboration, such as editors, spreadsheet tools, web browsers, multimedia players, and visual conference tools. Finally, compared to high-end computing systems that are often built to last for years, desktop workstations at research institutions get updated more often and typically have higher compute power than individual nodes of a large, parallel system. This is especially advantageous for running sequential programs, and there exist many essential scientific computing tools that are not parallel. Applications that were once beyond the capability of a single workstation are now routinely executed on personal desktop computers. The combination of fast CPU, large memory, and the prospering Linux environment provides scientists with a familiar—yet powerful—computing platform right in their office, often times enabling them to avoid the overhead of obtaining parallel computer accounts, frequent data movement, and submitting, as well as waiting for the completion of batch jobs.

While personal computers are up to their important roles in scientific workflows with advantages in human-computer interface and processing power, *storage* nowadays usually becomes their limiting factor. Commodity desktop computers are often equipped with limited secondary storage capability and I/O rates. Shared storage in university departments and research labs are mostly provided for hosting ordinary documents such as email and web-pages, and usually comes with small quota, low bandwidth, and heavy workloads. This imbalance between compute power and storage resources leaves scientists with two unattractive choices when processing datasets larger than their workstations' available disk space. First, their workstations can remotely access the data sets—but the wide-area network latencies kill performance. Second, they can use a high-performance computer, which has sufficient disk space–but will have to perform their computation either at a crowded head node or through a batch system.

Users may also choose to install a large storage system accessible from their desktop workstations. However, this is not cheap. Although disks themselves are relatively affordable today (at \$1000 to \$2000 for 1 TB), building a storage system requires expensive hardware such as fiber channel switches. For example, a 365GB disk array currently costs over \$6000 and a 4TB array costs over \$40,000,[1] which is a non-trivial expense, especially for academic and government research environments. This has not yet taken into account the maintenance costs. Although price is expected to fall for the same capacity, data size is expected to rise, often at a higher speed. In fact, parallel simulations can easily generate TBs of data per application per day already [Bair et al. 2004]. When groups of users store their scientific datasets in a shared storage system, even a large space can quickly be exhausted, as demonstrated by shared scratch file systems at supercomputer centers.

Further, even when workstation-attached storage is abundant, users normally choose not to retain copies of the downloaded scientific datasets on their desktops beyond the processing duration. These datasets are several orders of GB or larger and are usually archived in mass storage systems, file servers, *etc.* Subsequent requests to these datasets involve data migration from archival systems at transfer rates significantly lower than local I/O or LAN throughput [Lee et al. 2002; Lee

[1]Price quote from www.ibm.com as of 2005.

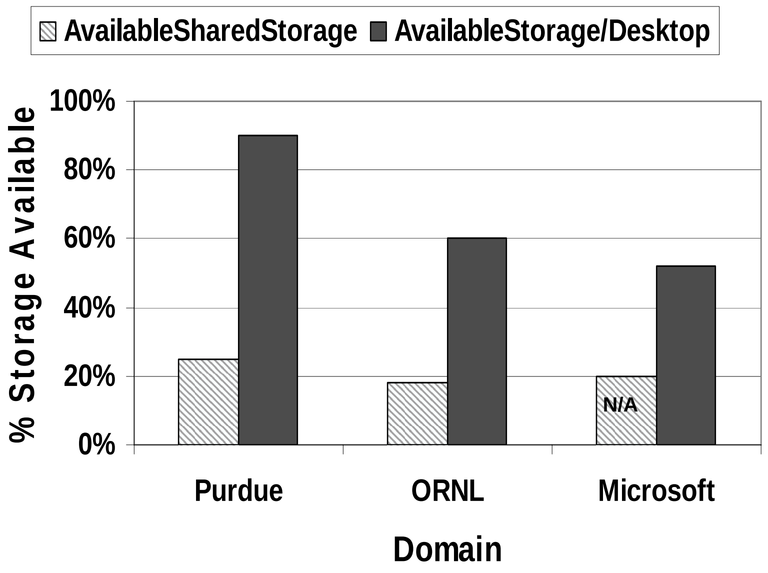


Fig. 1. Space availabile on shared storage servers and desktops based on surveys at Microsoft, Purdue and ORNL. Surveys included shared servers at Purdue and ORNL amounting to several TB. Data from Microsoft was not available for this category. Desktop surveys included several hundreds of workstations at both Purdue and ORNL and at least 50000 machines from Microsoft. Desktop storage capacity typically ranged from 30 to 80GB of space per workstation.

et al. 2004; Vazhkudai and Schopf 2003].

Meanwhile, collectively a large amount of disk space remains idle on personal computers within academic or industry organizations (Figure 1). Studies show that on average, at least half of the disk space on desktop workstations is idle, and the fraction of idle space increases as the disks become larger [Adya et al. 2002; Douceur and Bolosky 1999]. In addition, most workstations are online for the vast majority of the time [Chien et al. 2003]. A desirable low-cost alternative then, is to **harness the collective storage potential of individual workstations** much as one harnesses idle CPU cycles [Litzkow et al. 1988]. 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 2), 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 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 download these terabytes of result-data onto local

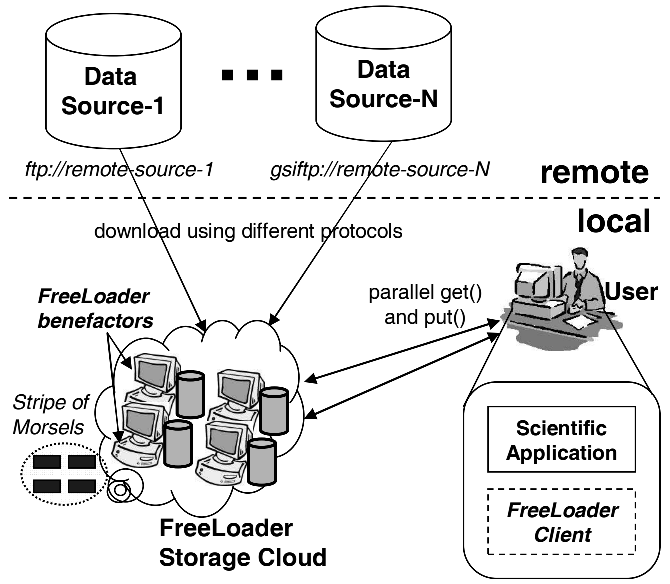


Fig. 2. Envisioned FreeLoader Environment

machines and use visualization tools to study them 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."

It is important to note that interactive data-intensive computing has several unique characteristics.

—Datasets are usually write-once-read-many. Further, they are usually shared since people within the same organization, *e.g.*, a research group or academic department, often times have shared interest on certain datasets [Otoo et al. 2004].

—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 [ncbi 2005; The Astrophysical Research Consortium 2005; Szalay and Gray 2001].

—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 in the collaboration [Iamnitchi et al. 2004]. 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.

Although there exists other work on desktop storage aggregation [Adya et al. 2002; Butt et al. 2004], FreeLoader is novel in the following aspects. First, rather than providing a general-purpose distributed file system, FreeLoader is designed to handle transient uses of bulk scientific data. 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. As a result, FreeLoader provides high-performance I/O using only donated idle storage without adversely impacting the native workload on the donated machines. Finally, FreeLoader derives several best practices from different classes of storage systems. For instance, it adopts: desktop storage scavenging from P2P systems; data striping and parallel I/O from parllel file systems; and caching from cooperative caching systems. It combines the aforementioned key strategies and several others to build a collaborative caching system based on storage contributions and applies it to an emerging problem space of bulk scientific data accesses and sharing.

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 [Strickland et al. 2005].

The rest of the paper presents the design, implementation, and evaluation of our FreeLoader prototype. Section 2 discusses related work. Section 3 presents the overall architecture design of FreeLoader, and Section 4 gives more details on the implementation of our FreeLoader prototype. Performance and impact experiment results are discussed in Section 5. Section 6 concludes the paper.

2. RELATED WORK

Tens of networked and distributed file systems exist as shared storage (*e.g.*, NFS [Nowicki 1989]), LOCUS [Popek and Walker 1985], AFS [Howard 1988; Morris et al. 1986] , CODA [Satyanarayanan et al. 1990], 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 [Adya et al. 2002] and Kosha [Butt et al. 2004]). However, all the above systems serve as file systems and target general-purpose file system usage patterns. Additionally, GFS [Ghemawat et al. 2003] is a distributed file system designed for data-intensive tasks, but is proprietary, uses dedicated disks, and is specialized for Web searches. In contrast to these existing file 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. Further, FreeLoader treats the aggregated scavenged space as a cache, constantly uploding or evicting datasets based on client access patterns.

Parallel file systems (*e.g.*, GPFS [Schmuck and Haskin 2002], Lustre [Cluster File Systems, Inc. 2002], PVFS [Carns et al. 2000]), Frangipani [Thekkath et al.

Systems	**FS**	**Cache**	**Striping**	**Scavenging**	**WAN**
FreeLoader	No	Yes	Yes	Yes	No
Parallel file systems					
GPFS, Lustre, PVFS, Frangipani+Petal	Yes	No	Yes	No	No
Distributed file systems					
Zebra	Yes	No	Yes	No	No
NFS, AFS, Coda	Yes	No	No	No	Yes
Google FS	Yes	No	No	No	No
FARSITE	Yes	No	No	No	No
IBP+exNode [Beck et al. 2002]	No	No	Yes	No	Yes
P2P Storage					
Kosha	Yes	No	No	Yes	No
BitTorrent	No	No	Yes	No	Yes
Gnutella, Kazaa, Freenet	No	No	No	Yes	Yes
Squirrel	No	Yes	No	No	No
Cooperative caching					
xFS, GMS, hints	No	Yes	No	No	No

Table I. Comparison with related file and storage systems. The column "FS" indicates whether a system is designed to present general-purpose 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 striping. "Scavenging" indicates the use of space contributions either in part or whole. The "WAN" column denotes use as wide-area storage.

1997] and Petal [Lee and Thekkath 1996], target large datasets, provide sustained high I/O throughput, and are tightly integrated with supercomputers. Recently, distributed logical disks [Frolund et al. 2003] are being built from a decentralized collection of commodity storage appliances, extending ideas from Petal with replication, volume management and load balancing. Such systems handle replication at a disk segment level and not at a dataset level. Aforementioned systems are widely used by FreeLoader's target users: 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 [Hartman and Ousterhout 1995]), it is usually done in a uniform or symmetric way with relatively homogeneous settings of these systems. FreeLoader explores overlapping network data transfer and local I/O with a novel *asymmetric striping* technique.

The striping and I/O bandwidth aggregation in FreeLoader is similar to systems like Swift [Cabrera and Long 1991] that address the data rate mismatch between application needs and storage systems through parallel I/O across the network. However, our system is different in its attempt to provide a parallel I/O interface across scavenged unreliable desktop storage that poses new opportunities and challenges for striping. In addition, FreeLoader treats the entire scavenged space as a cache unlike Swift, which is designed as a bandwidth aggregation system.

FreeLoader can be viewed as cooperative caching [Dahlin et al. 1994; Feeley et al. 1995; Sarkar and Hartman 1996; Gadde et al. 1998] 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. Further, the primary goal of cooperative caches is to enable access locality and sharing. The performance benefit is due to a hit in the local cache that obviates the need for wide-area accesses. In FreeLoader, this performance gain, however, is amplified manifold due to the parallel retrieval of the striped, in-cache dataset.

Also related is Batch–Aware Distributed File System (BAD–FS [Bent et al. 2004]), which constructs a cooperative cache/scratch environment from storage servers or appliances [Bent et al. 2002], specifically geared towards I/O intensive batch workloads in a wide-area environment. While we can draw parallels with FreeLoader in the sense of catering to data–intensive workloads and enabling caching, we differ significantly in the sense that FreeLoader's cache environment is based on very loosely connected desktop storage as opposed to BAD-FS's storage appliances.

Finally, multiple large scale P2P [Crowcroft and Pratt 2002] systems exist (*e.g.*, Gnutella [Markatos 2002], Kazaa [SHARMAN NETWORKS, Inc. 2005], Freenet [Clarke et al. 2000] and BitTorrent [Cohen 2003]). PAST [Druschel and Rowstron 2001], CFS [Dabel et al. 2001], Ivy [Muthitacharoen et al. 2002] and OceanStore [Kubiatowicz et al. 2000] facilitate wide-area distributed data storage by providing persistence and reliability. The Shark distributed file system [Annapureddy et al. 2005] attempts to scale centralized (NFS–like) servers through the use of cooperative caching and peer-to-peer distributed hash tables (DHTs [Ratnasamy et al. 2001]). Also akin to our approach is Squirrel [Iyer et al. 2002], 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 have usually been designed for *content sharing*, while FreeLoader has additional goals of *space aggregation* and *bandwdith aggregation*. BitTorrent and Shark aggregate bandwidth as well. 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 [Otoo et al. 2004], differs significantly from that of P2P file sharing systems [Gummadi et al. 2003], 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

3.1 Assumptions

In this section, we highlight some of our design choices. Before we present FreeLoader's architecture, we first list below design issues regarding system scope and assumptions.

Scalability: Our target storage resource management environment is intended to support tens to thousands of workstations within an administrative domain, handling data access requests from numerous clients as well.

Connectivity and Security: We assume a well connected corporate LAN setting but not high–speed communication environments expected by parallel file systems. The implementation described in this paper does not provide any security mechanisms. There are plenty of existing studies on secure distributed storage using untrusted components (*e.g.*, [Adya et al. 2002]). Many of these mechanisms can be leveraged by FreeLoader. We examine this cost in Section 5.2.2

Heterogeneity: User desktop workstations come in all flavors ranging from operating system diversity to machine characteristics, CPU speeds, disk speeds, network bandwidths to varying temporal loads. Our architecture will need to accommodate such a diverse mix and exploit the performance differences therein.

Scientific Data Properties: FreeLoader is intended for accessing scientific datasets, rather than general-purpose documents and data that people prefer to store in secure, high-reliable file systems. As mentioned in Section 1, scientific datasets are typically large, immutable, accessed in sequential manner, and almost always with a primary copy archived in a remote storage system.

User Impact Control: FreeLoader is based on space contribution from individual users and revolves around the premise that a user will ultimately withdraw contribution if overly burdened. Thus, our design needs to reflect this guiding principle with support for performance impact control, so that the slowdown incurred by FreeLoader on space donors' native workloads can be tuned.

Next we present the overall architecture of FreeLoader by describing its major components.

3.2 FreeLoader Components

FreeLoader aggregates donated storage into a single storage system. The basic architecture consists of two components. The *management* component maintains the metadata and performs high-level operations, such as replication and cache replacement. The *storage* component consists of *benefactor* nodes that donate space along with I/O and network bandwidth. Data storage and retrieval are initiated by the *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.

3.2.1 *Management Component.* The management component maintains metadata (such as dataset names) and performs lookup services to map a client-requested dataset to morsels on benefactors. This component does not touch any data in the dataset. Because the amount of metadata is significantly smaller than real data, the management component 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 [Ghemawat et al. 2003]. Clients communicate with a manager node to obtain morsel mapping, then directly contact the benefactors for morsel transfer.

Besides morsel location lookups, the management component stores client-specific metadata for added functionality, such as per-morsel fingerprint checksums for increased dataset integrity. Such services, including encryption and decryption, are optional client-side filters that have little storage overhead at the management component. Moreover, the computational costs are paid by clients (not benefactors) who elect to use them.

For aggregating I/O bandwidth, FreeLoader adopts software striping [Hartman and Ousterhout 1995] 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 in FreeLoader's heterogeneous and dynamic settings 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, the user who imports or creates a dataset is likely to visualize or analyze it most often. Recognizing this, 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 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 *reclaimed space* (withdrawn by the benefactor), 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.

3.2.2 *Storage Component.* The storage component, which runs on benefactor nodes, manages all the morsels in the system. The primary function of this component is servicing *get* and *put* morsel requests. Since FreeLoader stores read-only datasets and accesses to scientific datasets have temporal locality [Otoo et al. 2004], get requests will dominate traffic.

A benefactor node is an ordinary user machine that has donated certain idle disk space and has installed the benefactor component of FreeLoader 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 send heartbeat or "I'm alive" messages to the manager(s).

Another important task of the storage component is performance impact control on benefactors' native workload. 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 [Anderson et al. 2002; Litzkow et al. 1988]. Such a strategy is not only overly conservative [Gupta et al. 2004; Novaes et al. 2005], but also infeasible for FreeLoader as it incurs 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 [Strickland et al. 2005].

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.

This high-level impact control will further be complemented by benefactors' local impact control, which is done by performance impact benchmarking, real-time monitoring of the native workload's resource consumption, and throttling the FreeLoader daemon's execution. This method can contain the actual performance

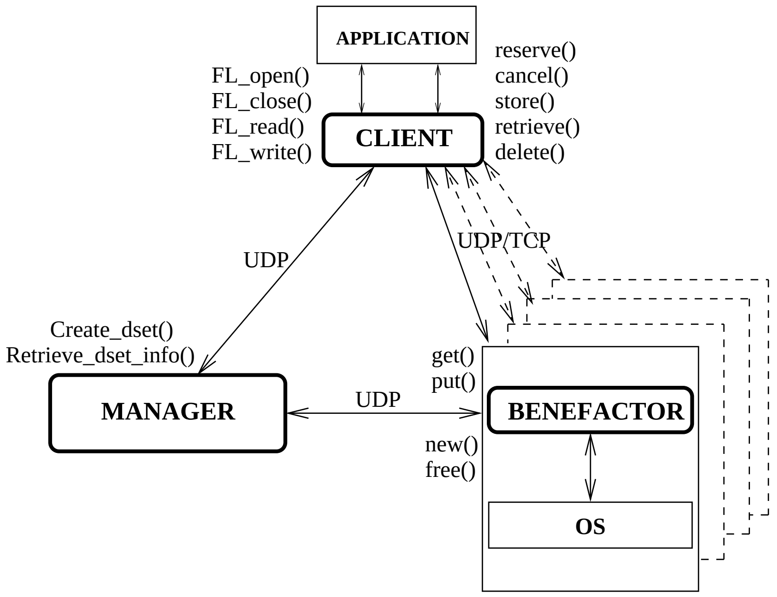


Fig. 3. Modules and interfaces in prototype

impact on native workloads within a pre-specified threshold. Interested readers are referred to our paper discussing the benefactor-side performance impact control [Strickland et al. 2005]. Section 5 demonstrates our empirical performance impact study and control through striping.

4. PROTOTYPE DESIGN AND IMPLEMENTATION

Our FreeLoader proof-of-concept prototype implements major functionalities described in Section 3. Figure 3 shows its main modules as well as the interfaces between them. This prototype verifies the following rationales.

—Harnessing workstation storage delivers aggregate data retrieval rates at least comparable to those currently possible accessing existing local higher-end storage systems, and significantly higher than those accessing remote systems.

—Software striping delivers both high aggregate data access throughput and scalability with regard to stripe width in a LAN environment. In particular, with our asymmetric striping technique, a client can efficiently combine its network data transfer with local disk I/O.

—Data serving activities exert a tolerable impact on workstation's native workloads and this impact is controllable by throttling the data transfer rates.

—The overhead of the FreeLoader framework, notwithstanding bulk data transfer, is acceptably low and reduces as stripe width increases.

4.1 Manager Design and Implementation

The FreeLoader architecture and protocol design allows for multiple managers for increased system performance and availability. It has been demonstrated by large commercial distributed systems such as the Google File System that a single manager can successfully provide metadata management and request distribution support for thousands of storage nodes [Ghemawat et al. 2003]. Our prototype implementation is based on a single manager and we will focus on this simplification in the rest of our discussion. 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.

4.1.1 *Metadata Management.* 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.

Also part of the manager metadata is information regarding namespace maintained as directory metadata. Datasets are stored into FreeLoader space by having the clients specify the location/protocol of the remote primary copy (URI). The URI is maintained by the manager as part of metadata for each dataset as its identifying tag in the FreeLoader namespace.

4.1.2 *Data Placement.* The manager has to decide where to place each incoming dataset. As mentioned in Section 3.2.1, software striping is adopted due to its two-fold benefits in FreeLoader: increasing the client-side aggregate data access bandwidth and reducing the benefactor-side performance impact.

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.

Given a dataset with specified stripe width and stripe size, the manager uses a striping algorithm to select a subset of benefactors to store it. An intelligent striping algorithm should manage space efficiently while also factoring in performance capabilities of benefactors. We have implemented three data striping strategies: *round-robin* to benefactors excluding the client that stores the dataset, *predictive* that factors in capability differences between benefactors during striping, and *asymmetric* that stripes (unevenly) to both benefactors and the client (called *host client* of the dataset in question).

Round-Robin striping: Note that with software striping, "stripe width" may differ with "the number of benefactors a dataset is striped onto" (we call these benefactors the *stripe node set* of that dataset). The former refers to the number of benefactors the client will be transferring data to or from *simultaneously*. For example, with a stripe width of 4, the first half of a dataset maybe striped onto benefactors 1, 2, 3, and 4, while the second half striped onto benefactors 5, 6, 7, and 8, without affecting the overall space requirement or the client perceived data transfer rate.

However, for practical purposes, it helps to maintain a small stripe node set. First, since benefactors are managed individually and may be down or very busy at any time, as the stripe node set grows, the chance of a dataset missing part of its morsels also grows significantly. Second, a larger stripe node set implies that on average more datasets will be stored on each benefactor, increasing the cost in metadata management due to data relocation when benefactors quit or reclaim spaces from FreeLoader. In particular, a striping strategy that maximizes the stripe node set size will have a large number of datasets locked and unavailable during such data relocation processes. Such a strategy will also cause benefactors with higher space contributions to be accessed more often, with a load proportional to space contribution. Finally, having a large node set increases the overhead of adding, migrating, or removing datasets, and increases the burden on the manager. This is because a larger number of benefactors will be involved in metadata and file data updates, with the startup overhead less amortized over communicating morsel-level information.

Therefore, we first consider a striping strategy that would fix the stripe node set size at a given stripe width. Below we demonstrate that even with this preset stripe width and the simple round-robin striping strategy, to optimize *where* to place specific stripe units in a heterogeneous environment is a difficult problem. In FreeLoader's target scenarios, benefactors come with varied space availability, and an optimal placement algorithm should fit as many datasets as possible to these benefactors.

This 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 show that a known NP-hard problem, *Minimum Bin Packing* [Garey and Johnson 1979], can be reduced to the off-line version of this Stripe Fit problem. Below is the proof.

Definition 4.1. *Given a finite set U of items, a size $s_u \in Z^+$ for each $u \in U$, and a positive integer bin capacity C, a solution to the* **Minimum Bin Packing** *problem is a partition of U into the minimum number of disjoint sets U_1, U_2, ..., U_m, such that the sum of the item sizes in each U_i is C or less.*

Definition 4.2. *Given a sequence of n datasets $D = d_1, d_2, ..., d_n$, a size s_d and a stripe width w_d for each $d \in D$, a set of m disks $K = \{k_1, k_2, ..., k_m\}$, and a capacity S_k for each $k \in K$, a solution to the* **Stripe Fit** *problem is a dataset striping plan that maximize the length of D', a prefix of D to store in K, such that each dataset d in D' is divided into w_d equal partitions and stored in w_d disks in K, where the sum of dataset sizes in each k is S_k or less.*

Theorem 4.1. *The Minimum Bin Packing problem can be reduced to the Stripe Fit problem.*

Proof of Theorem 4.1. Given an instance of the Minimum Bin Packing problem, we show that this problem instance can be reduced in polynomial time into an instance of the Stripe Fit problem.

A function f takes the input of the Minimum Bin Packing problem $< U, C >$, and outputs a Stripe Fit problem $< D, K >$, where

—$|K| = n = |U|$, where for each $k \in K$, $s_k = C$.

—$D = d_1, d_2, ..., d_n, d_{n+1}, d_{n+2}, ..., d_{2n}$. For $i \in [1, n]$, $s_{d_i} = s_{u_i}$, while for $i \in [n+1, 2n]$, $s_{d_i} = C$.

—$w_{d_i} = 1$, for all $d_i \in D$.

Essentially, f maps a Minimum Bin Packing problem to a Stripe Fit problem, by creating $n = |U|$ uniform-sized disks, and $2n$ datasets to be striped to these disks with a stripe width of 1. The first n datasets have the same sizes as items in U, while the second n are "dummy datasets" who can each fill a bin of capacity C.

A function g takes the solution to the above Stripe Fit problem instance, l, where l is the length of the maximum prefix of D that can be striped to K, and maps it to m, the minimum number of bins that can accomodate U. A total of n disks are used to store the first l datasets in D, where the last $l - n$ datasets in this prefix are all "dummy datasets" of size C, the uniform disk capacity. Therefore, the first n datasets occupy $n - (l - n) = 2n - l$ disks, and $m = g(l, n) = 2n - l$. Note that $n \leq l \leq 2n$.

m is the mininum number of bins that can hold U. Otherwise, there exists $m' < m$, and a partition of U to fit in m' bins. Since this partition also gives a striping plan of $d_1, d_2, ..., d_n$ to m' disks with capacity C using a stripe width of 1, by substituting the striping plan of the first n datasets in the Stripe Fit problem solution above, one can store $m - m'$ more (dummy) datasets from D in K, contradicting with that l is the optimal answer.

It is easy to see that both f and g can be computed in polynomial time. □

The above shows that the offline complexity for finding an solution that optimizes space utilization while forcing each dataset to be striped onto a preset number of benefactors. FreeLoader has to make on-the-fly decisions as datasets arrive. Therefore we relaxed the requirement that the size of the stripe node set has to equal the stripe width, and designed a greedy algorithm to maximize the use of available space.

With this greedy algorithm, 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. If there does not exist enough benefactors with available space to sustain the stripe width w_i, we decrement the stripe width to $w_i - 1$, and repeat the process until the entire dataset is stored. This way, we automatically perform load balancing between benefactors, ensure that each dataset is accessed simultaneously from no more than w_i benefactors, and control the stripe node set size with each dataset.

Predictive Striping: While our greedy round-robin striping adapts to varying space availability on the benefactors, it does not factor in the potential heterogeniety in the participating benefactor nodes. For instance, the benefactors can have varying connectivity (GigE, 100Mb/sec, etc.), different I/O rates, processing speeds, transient loads, resulting in disparate morsel serving rates. To address and exploit this rate difference, we build a predictive striping strategy. With this technique, the morsel distribution per benefactor is commensurate to its predicted transfer rate, decided based on a previous history of benefactor-to-client morsel transfers. Thus, depending on previous transfer rates, faster benefactors will get assigned larger portions of data and vice-versa.

To obtain a rate estimate, benefactors log and update the manager on their transfer rate to the client. The manager can use this history and derive an average or last observed throughput as morsel delivery estimate. The manager chooses a width of nodes, w, based on space availability and morsel delivery rates. It then generates a morsel distribution map, dividing the dataset into morsels, corresponding to w benefactors. Each benefactor, $i, 1 \leq i \leq w$, has an estimated morsel transfer rate of B_i to the client. In theory then, the aggregate bandwidth achievable by the client for the entire download is:

$$A = \sum_{i=1}^{w} B_i,$$

where A is the aggregate throughput and B_i is the estimated morsel transfer rate per benefactor.

Assuming the client is capable of handling the bandwidth surplus, morsel distributions are calculated as follows. For each benefactor $i, 1 \leq i \leq w$, and for a dataset size, S, the number of morsels per node is:

$$s_i = \frac{B_i S}{A},$$

where s_i is the number of morsels per benefactor. Thus, the number of morsels per benefactor is commensurate to its transfer rate and its ratio of contribution to the achievable aggregate bandwidth. Faster benefactors are assigned to deliver bigger portions of the dataset, while slower benefactors are assigned smaller pieces.

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 afterward. 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 Design and Implementation

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 3 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, as shown in Figure 3. 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.

In addition, the morsel service layer also provides an *import* interface with which benefactors upload datasets from remote sources using the location/protocol mentioned in the URI. The client, after obtaining morsel maps from the manager, uses this interface to specify the URI from which to import the dataset, to a stripe width of benefactors. Our current implementation supports imports through several protocols such as wget and hsi.

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 Design and Implementation

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).

4.3.1 *Client-side Buffer Management and Data Access Interfaces.* 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").

Besides whole-file gets and puts between the FreeLoader space and a client's private disk space, we have 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. Such interfaces are especially useful when users use FreeLoader as a transparent scratch space of large datasets that do not fit into their workstations' local storage. 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 5.3 demonstrates a widely-used application using these interfaces in processing a biological sequence database stored in FreeLoader.

4.3.2 *Client-side Filters.* Storing and sharing scientific data within administration boundaries seldom raise much security or data integrity concern. However, in certain cases clients might require some security features to provide additional protection to their data on untrusted workstations. To address this, we implemented several features such as data checksumming and encryption as client-side filters in our FreeLoader prototype. These are supported as optional features that can be enabled by clients who are willing to pay the additional cost involved with the filter. Note that these filters do not add any computation burden to the benefactors, and require only very limited extra storage at the manager in terms of metadata on checksums and encryption keys.

For checksumming, the client computes per-morsel checksums and store them as a part of dataset metadata with the manager. Currently checksums are calculated for each morsel as a 128-bit value using the MD5 message-digest algorithm [Rivest 1992]. Since we choose morsel sizes at the MB level, this only takes an additional space of 8 bytes per morsel. For very large datasets, morsels can be further combined to reduce this cost. Upon subsequent morsel retrieval, new checksum values are generated and compared with the values from the manager.

For encryption, morsels are encrypted using DES encryption. A 64-bit key is generated and stored along with dataset metadata prior to a morsel put. Upon retrieving a dataset, authorized clients decrypt morsels using the same key.

In our target scenarios, datasets are write-once-read-many. Therefore, we use our prototype to verify the impact of performing such additional client-side operations on the client data retrieval rates (as before). For both filters, the computation is overlapped with data communication to pipeline morsel retrieval with checksum verification or decryption. There is little CPU contention and the cost depends on how expensive the filter computation is. Our results (discussed in Section 5.2.2) demonstrate mixed results: the reduction in retrieval rates is very small for decryption, but significant for checksum verification.

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 4) 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 4 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 configuration: Dual Pentium III, Linux 2.4.20-8 kernel and 100 Mb/sec Ethernet. The benefactor configuration is intended to portray the worst case scenario and study FreeLoader performance in the face of sub-optimal configurations. (2) The PVFS [Carns et al. 2000] parallel

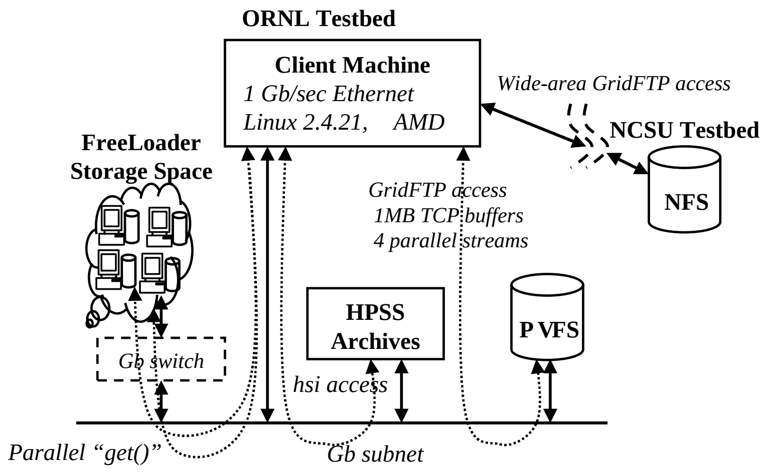


Fig. 4. FreeLoader testbed

FreeLoader	15 Benefactors, 1 Manager and 1 client
(Data set size)	256MB to 64GB
(Morsel size)	1MB
(Stripe size)	1MB, 2MB, 4MB, 8MB, 16MB, 32MB
(Stripe Width)	1, 2, 4, 6, 8, 9, 10, 12
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 at HPSS
(Cold)	Fetch forced from tape at HPSS
NCBI	wget from `http://www.ncbi.nlm.nih.gov`

Table II. Throughput test setup

file system on the ORNL TeraGrid Linux cluster outside ORNL firewall with several terabytes of storage (accessed through GridFTP [Bester et al. 1999]).(3) The HPSS [Coyne and Watson 1995] archival storage system 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 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. The client machine also uses tools such as *globus_url_copy* [Bester et al. 1999] to access the GridFTP server at the PVFS, hierarchical storage interface (hsi) client access to HPSS and FreeLoader parallel retrieve client. (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 Data Retrieval 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 [ncbi 2005]) frequently used by scientists. We conducted several transfers experiments over a week (see summary in Table II) and report average results.All data retrieval performance is measured from the client box. For FreeLoader, we tested different striping techniques and several combinations of stripe sizes and stripe widths. For PVFS and NFS, we used GridFTP with tuned TCP buffer settings and parallel streams to account for the "wizard-gap." For HPSS, we tested both "hot" and "cold" accesses to factor in the disk cache effect. Each file transfer was repeated several times and our results show the average.

Figure 5 compares the "best of class" performance using FreeLoader and other data sources. Figure 5 uses asymmetric striping (client + 8 benefactors) for FreeLoader. With asymmetric striping, we used a 3:8 morsel ratio for client:benefactors—i.e., three morsels on the client and one morsel on each of the eight benefactors in every stripe cycle. 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. 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 in storing and accessing their datasets.

5.2.1 *Effect of Striping on FreeLoader Performance.* With reference to striping, two things are of interest—stripe size and width. Stripe size delves into how many morsels are stored/retrieved to/from each benefactor, while stripe width deals with the number of benefactors across which the striping is performed. We study these two variables for several data set sizes. Figures 6(a) and 6(b) depict the effect of varying stripe sizes for different dataset sizes, with the stripe width fixed at 4 and 2 benefactors. They show no substantial throughput improvements in varying stripe sizes. This result can be instrumental in choosing suitable buffer sizes at the client end for assembling striped data. For instance, the client allocates a buffer of size $stripewidth \times (stripesize + 1)$ morsels in retrieving data, as discussed in Section 4.3). A smaller stripe size means smaller client-side memory requirement.

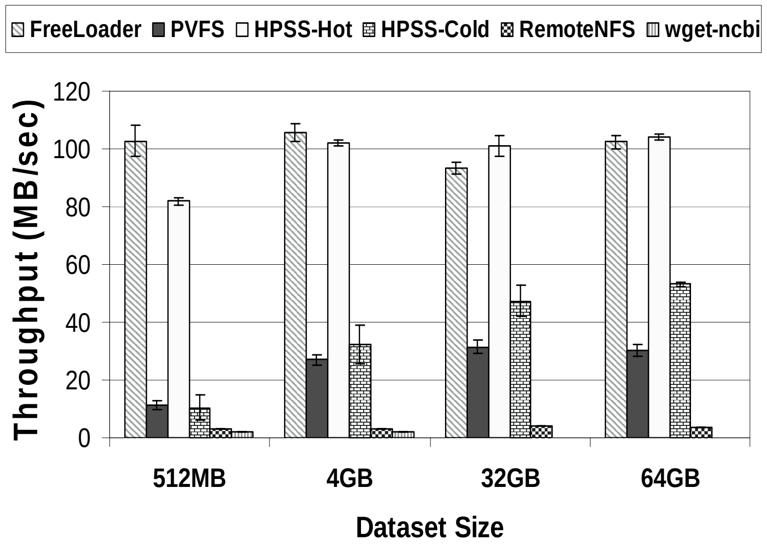


Fig. 5. "Best of class" comparison of data retrieval throughput, with 95% confidence ranges. FreeLoader throughput is an asymmetric striping on eight benefactors. wget-ncbi datasets were unavailable for larger sizes. In the rest of experiments, FreeLoader's performance variance is similar to that depicted in this figure and error bars are omitted.

Figure 7 denotes the effect of varying the stripe width for different data sizes, with the stripe size fixed at 1MB morsel. By increasing the stripe width, we see almost a linear throughput improvement until the client reaches saturation, due to better utilization of the residual bandwidth available at the client and extra I/O bandwidth. In these experiments, we noticed that the client machine saturates around 12 benefactors for all dataset sizes and stripe sizes. Thus, adding more benefactors—from then on—offers no further gain.

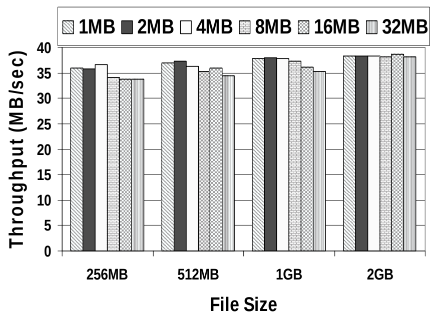
To study the effect of a heterogeneous set of benefactors on client perceived throughput, we assemble a mix of donor nodes with 100Mb/sec and 1Gb/sec Ethernet connectivity. Our test comprised a round-robin striping of a 1GB dataset across a width of nodes ranging from an entirely homogeneous mix (all Gb/sec machines) to the gradual induction of one of more 100Mb/sec machines. Previously, in Figure 7, we studied a homogeneous mix comprising entirely of 100Mb/sec benefactors and observed that, for our testbed, the client throughput saturated between 10 and 12 nodes. In this experiment (Figure 8 Round-robin striping), we start with all Gb/sec benefactors and observe that just three of those nodes (ratio 0:3 means no 100Mb/sec nodes) are sufficient to saturate the client. However, in a realistic setting, a given dataset cannot always be striped across all Gb/sec benefactors for the following reasons. First, Gb/sec nodes may not always be available and even when available, may not have enough space. Second, striping every dataset on well-connected nodes only, would introduce load imbalance. Third, a smaller stripe width might render larger portions of a dataset unavailable in case of node failure, which can be a recurring event in a desktop scavenging setting.

Benefactor ID	Connectivity	Morsel-serving Rate (MB/sec)
1	Gb/sec	17.64
2	Gb/sec	34.4
3	Gb/sec	44.11
4	100Mb/sec	10.07
5	100Mb/sec	10.07
6	100Mb/sec	10.59
7	100Mb/sec	10.59
8	100Mb/sec	10.06
9	100Mb/sec	10.59
10	100Mb/sec	11.17
11	100Mb/sec	10.5
12	100Mb/sec	11.1

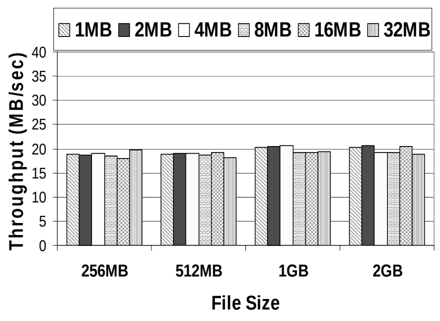
Table III. Benchmark throughput, average in MB/s, for different benefactors for 1GB transfers

For these reasons, we study widening the stripe width and the induction of more benefactors in Figure 8. A larger stripe width, as we will illustrate later, has the desired effect of reducing the impact on participating nodes. We found that introducing another Gb/sec node only contibuted to a reduction in throughput due to client saturation. Adding 100Mb/sec nodes, however, has an interesting effect. A 1:3 (one 100Mb/sec node and 3 Gb/sec nodes) mix results in a substantial dip in client throughput. This can be attributed to the fact that a simple round robin striping across a 1:3 mix, results in an equal distribution of morsels across the benefactors and does not factor in their different capabilities. Faster benefactors finish serving morsels to the client much earlier than their slower (100Mb/sec) counterparts and are idle, waiting on them to finish. However, as we gradually induct more 100Mb/sec nodes, the client throughput progressively increases until saturation as observed at a 7:3 ratio. This test suggests that data placement techniques factoring in client and benefactor heterogeniety can significantly improve client throughput and better utilize available nodes.

Our next test comprised factoring in benefactor heterogeniety, namely their connectivity. We use each benefactor's morsel serving rate as a distinguishing element in the data striping policy. To this end, we benchmarked each benefactor's morsel serving rate to the client using a training set of several 1GB transfers. Results are summaried in Table III. We then used these benchmark numbers from each benefactor as a measure of the rate they can deliver. Consequently, we place on each benefactor morsels, commensurate to this predicted throughput. Figure 8, Predictive striping, summarizes our results. From the graph, we can see how the striping strategy can maintain a desired width and yet stripe morsels based on delivery rates. Assuming space is no object, a 1:3 mix in this case, will stripe 162, 324, 419 and 95 morsels on the first four benefactors in Tabel III, as opposed to 250 morsels on each (as in round-robin). Since slower benefactors get assigned smaller number of morsels, faster benefactors are used judiciously for morsel delivery rather than waiting on slower benefactors to finish. Such a distribution can help exploit the capability differences between the benefactors and aid in stricking a balance



(a) Stripe size variation (stripe width=4 benefactors)



(b) Stripe size variation (stripe width=2 benefactors)

Fig. 6. Stripe Size Variation

between spreading the load and achieving better throughput.

Asymmetric striping is another form of exploiting heterogeniety between client and benefactor configurations. 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 9(a) and 9(b) 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 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.

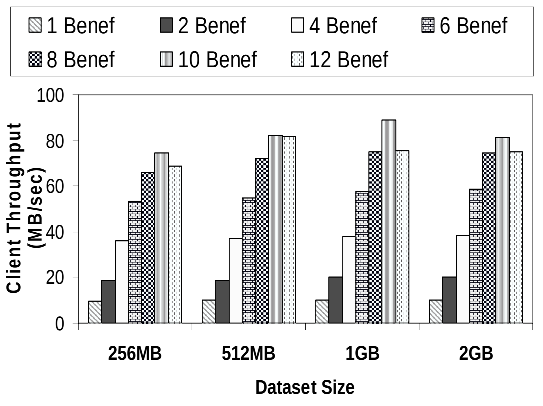


Fig. 7. Stripe width variation with round-robin striping

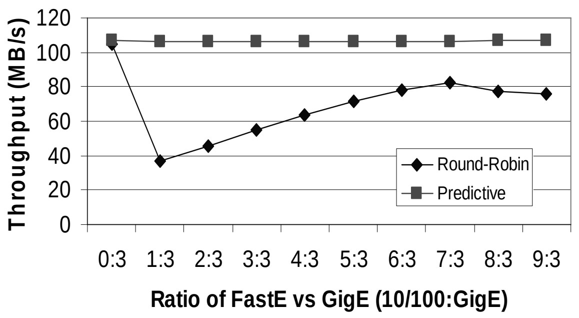
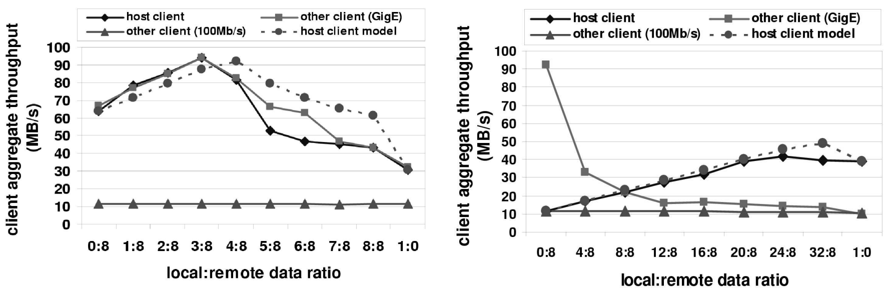


Fig. 8. Effect of a heterogeneous mix of benefactors on client throughput.



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

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

Fig. 9. FreeLoader asymmetric striping results

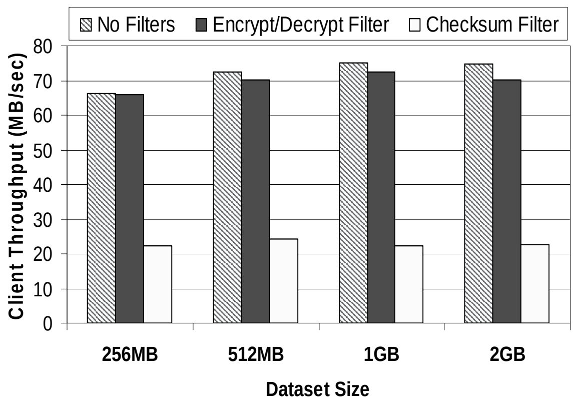


Fig. 10. FreeLoader Client throughput with and without checksum and encrypt/decrypt filters

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 9(a) and 9(b)). The GigE client, however, benefits from the GigE host client serving more data and shows a similar rate curve as the host client (Figure 9(a)). On the other hand, it experiences severe throughput drop as more data gets stored on the 100Mb/s host client (Figure 9(b)). 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.

5.2.2 *Client-side Filters.* Figure 10 compares the client throughput with and without the checksum verification and decryption filters for a variety of dataset sizes. As mentioned earlier, we are primarily concerned with the impact on client retrieval rates due to the use of these filters since gets dominate puts in the FreeLoader target environment. Results denote that the impact is up to 4% and 68% reduction in client throughput due to decryption and checksum verifications respectively. The 68% decrease in overall client perceived throughput related to block checksum verification comes from the inherent costs attributed to the MD5 algorithm itself. To verify this, we observed that checksumming a dataset streamed from the FreeLoader space is only 16% more expensive than memory-to-memory checksumming and is 30% cheaper compared to checksumming the same file residing on the client's local disk.

Further, this cost is only paid by clients who elect to use these services and these filters do not burden the FreeLoader framework but for storing the limited amount of additional metadata.

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

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

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.

Table IV 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.

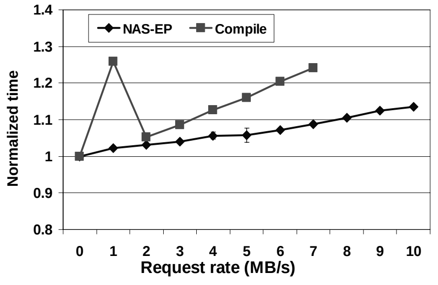
However, performance wise, *formatdb* is not an ideal application for FreeLoader, since it performs most of its input through the *fgetc* interface. This incurs high overhead, as FreeLoader has to perform morsel buffer look up for almost every byte read in. Therefore, this experiment allows us to observe the upper bound of FreeLoader's internal overhead.

5.4 Performance Impact on Benefactor Nodes

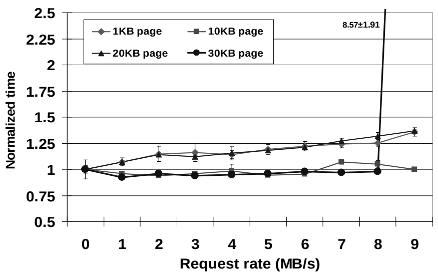
We have shown that FreeLoader can be an attractive storage choice from clients' view point. What about from space donors' view point? This section evaluates the performance impact on benefactors' native workloads, by measuring the slowdown factor of three typical types of activities: computation, network, 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

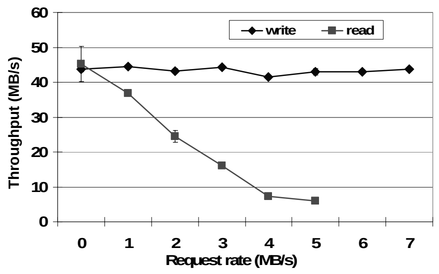
[2]http://www.nas.nasa.gov/Software/NPB/



(a) Computation



(b) Network



(c) Disk

Fig. 11. Benefactor impact results

completely CPU-bound, but represents typical computation-intensive user tasks. Figure 11(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 11(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 11(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 11 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 UN-zipping 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 12 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 slow-down 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 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 as plotted in the secondary y axis in Figure 12. This shows that striping serves as a means both to aggregate benefactor bandwidth and impact control. Again, more aggressive impact control can be performed at the benefactors

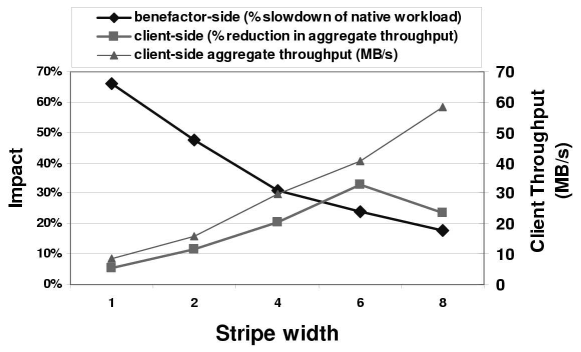


Fig. 12. High-level benefactor impact control by increasing the striping width. Primary y-axis plots benefactor slow down and client throughput loss ratio. Secondary y-axis plots actual client throughput.

[Strickland et al. 2005].

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. Based on our prototype and experiments, we verified the following rationales.

Our novel framework for aggregating idle, existing commodity storage resources complements high-end storage systems in caching large scientific datasets. The framework presents a scalable, layered architecture, comprising of benefactors, steered by managers, offering services such as reliability, performance and load balancing. We have presented a desktop storage scavenging system, FreeLoader, to construct a low-cost aggregate/shared storage cache that addresses growing application data demands. FreeLoader can be used for a variety of typical desktop processing, visualizations, diskless checkpointing, software update distribution and even incremental backups (albeit not without some quality of service) [Cox et al. 2002].

We validated distributed software striping in FreeLoader, and developed novel approaches to perform asymmetric data placement in order to optimize client achievable throughput. Based on this, we infer that FreeLoader can deliver high data retrieval rates for a low-cost scavenged storage, comparable to parallel and mass storage systems. We observed up to a threefold increase in throughput when compared against PVFS and HPSS cold accesses. Further, striping serves as an excellent technique to aggregate parallel I/O and balance the load among the benefactors. We observed, up to almost 10MB/sec speedup on retrievals with the addition of "one more benefactor" until client saturation.

The management overhead in maintaining/retrieving meta data, morsel distribution maps, etc., is significantly low. However, we suspect that with replication, reliability and security, the overhead will increase. Further, we have shown that additional features such as data encryptions and data integrity can be added as filters for clients willing to pay the cost, without penalizing other clients.

Finally, we measured the performance impact of storage scavenging on space donors' native workloads. We observed that the impact on the benefactor's native workload is minimal and throttling can help reduce it further. With CPU intensive operations afflicted by less than 1.5%, network latency less than 25% on Web transfers and typical user operations witnessing little to no impact, FreeLoader can offer substantial rewards for a low-cost storage system.

7. ACKNOWLEDGMENT

This work is supported in part by an IBM UPP award, the U.S. Department of Energy under contract No. DE-AC05-00OR2275 with UT-Battelle, LLC and Xiaosong Ma's joint appointment between NCSU and ORNL. The authors thank: Al Geist for supporting Nandan Tammineedi during the summer of 2004 through the HERE program at ORNL; Greg Pike, Stan White, John Cobb, Jamison Daniel, David Jung, Kasidit Chanchio, John Mugler, Jens Schwidder, Geoffroy Vallee and Ravi Madduri for helping with the testbed setup; and David Lowenthal for several useful comments on the paper.

REFERENCES

Adya, A., Bolosky, W., Castro, M., Chaiken, R., Cermak, G., J.Douceur, Howell, J., Lorch, J., Theimer, M., and Wattenhofer, R. 2002. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (Boston). Vol. 36. 1–14.

Anderson, D., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. 2002. SETI@home: an experiment in public-resource computing. *Communications of the ACM 45,* 11, 56–61.

Annapureddy, S., Freedman, M. J., and Mazieres, D. 2005. Shark: Scaling file servers via cooperative caching. In *Proceedings of 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI'05)* (Boston). 129–142.

Avery, P. and Foster, I. 2001. The griphyn project: Towards petascale virtual data grids. Tech. Rep. Technical Report GriPhyn-2001-15, `http://www.griphyn.org`.

Bair, R., Diachin, L., Kent, S., Michaels, G., Mezzacappa, T., Mount, R., Pordes, R., Rahn, L., Shoshani, A., Stevens, R., and Williams, D. 2004. Planning ascr/office of science data-management strategy. In *Department of Energy Office of Science Data Management Workshop* (Menlo Park).

Beck, M., Moore, T., and Plank, J. 2002. 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* (Pittsburgh). Vol. 32. 339–346.

Bent, J., Thain, D., Arpaci-Dusseau, A., Arpaci-Dusseau, R., and Livny, M. 2004. Explicit control in a batch aware distributed file system. In *Proceedings of the First USENIX/ACM Conference on Networked Systems Design and Implementation* (San Francisco). 365–378.

Bent, J., Venkataramani, V., Leroy, N., Roy, A., Stanley, J., Arpaci-Dusseau, A., Arpaci-Dusseau, R., and Livny, M. 2002. Flexibility, manageability, and performance in a grid storage appliance. In *Proceedings of the 11th High Performance Distributed Computing Symposium* (Edinburgh). 3–12.

Bester, J., Foster, I., Kesselman, C., Tedesco, J., and Tuecke, S. 1999. 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* (Atlanta). 78–88.

Butt, A., Johnson, T., Zheng, Y., and Hu, Y. 2004. Kosha: A peer-to-peer enhancement for the network file system. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (Pittsburgh). 51.

Cabrera, L.-F. and Long, D. D. E. 1991. SWIFT: Using Distributed Disk Striping To Provide High I/O Data Rates. *Computing Systems 4,* 4, 405–436.

Carns, P., III, W. L., Ross, R., and Thakur, R. 2000. PVFS: A Parallel File System For Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference* (Atlanta). 317–327.

Chien, A., Calder, B., Elbert, S., and Bhatia, K. 2003. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing 63,* 5, 597–610.

Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. 2000. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability* (Berkeley). Number 2009. 46–66.

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

Cohen, B. 2003. Incentives build robustness in bittorrent. In *First Workshop on Economics of Peer-to-Peer Systems* (Berkeley). 251–260.

Cox, L., Murray, C., and Noble, B. 2002. Pastiche: Making backup cheap and easy. In *Proceedings of the OSDI* (Boston). 285–298.

Coyne, R. and Watson, R. 1995. The parallel i/o architecture of the high-performance storage system (hpss). In *Proceedings of the IEEE MSS Symposium* (Monterey). 27–44.

Crowcroft, J. and Pratt, I. 2002. Peer to Peer: peering into the future. In *NETWORKING Tutorials*. 1–19.

Dabel, F., Kaashoek, M., Karger, D., Morris, R., and Stoica, I. 2001. Wide-area cooperative storage with cfs. In *Proceedings of the SOSP* (Chateau Lake Louise). 202–215.

Dahlin, M., Wang, R., Anderson, T. E., and Patterson, D. A. 1994. Cooperative caching: Using remote client memory to improve file system performance. In *Operating Systems Design and Implementation* (Monterey). 267–280.

Douceur, J. and Bolosky, W. 1999. A large-scale study of file-system contents. In *Proceedings of SIGMETRICS* (Atlanta). 59–70.

Druschel, P. and Rowstron, A. 2001. 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* (Chateau Lake Louise). 188–201.

Feeley, M. J., Morgan, W. E., Pighin, F. H., Karlin, A. R., Levy, H. M., and Thekkath, C. A. 1995. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mtn.). 129–140.

Frolund, S., Merchant, A., Saito, Y., Spence, S., and Veitch, A. 2003. Fab: enterprise storage systems on a shoestring. In *Proceedings of the Hot Topics in Operating System* (Lihue HI). 133–138.

Gadde, S., Chase, J., and Rabinovich, M. 1998. A taste of crispy squid. In *Proceedings of the Workshop on Internet Server Performance* (Madison). 129–136.

Garey, M. and Johnson, D. 1979. *Computers and Intractability: A guide to the theory of NP-completeness*, 4 ed. W. H. Freeman and Company.

Ghemawat, S., Gobioff, H., and Leung, S. 2003. The Google file system. In *Proceedings of the 19th Symposium on Operating Systems Principles* (Lake George). 29–43.

Gray, J., Liu, D., Nieto-Santisteban, M., Szalay, A., Heber, G., and DeWitt, D. 2005. Scientific data management in the coming decade. Tech. Rep. MSR-TR-2005-10, Microsoft.

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

Gummadi, P., Dunn, R., Saroiu, S., Gribble, S., Levy, H., and Zahorjan, J. 2003. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the 19th Symposium on Operating Systems Principles* (Lake George). 314–329.

Gupta, A., Lin, B., and Dinda, P. 2004. Measuring and understanding user comfort with resource borrowing. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing* (Honolulu). Vol. 00. 214–224.

Hartman, J. and Ousterhout, J. 1995. The Zebra striped network file system. *ACM Transactions on Computer Systems 13,* 3, 274–310.

Howard, J. H. 1988. An overview of the andrew file system. In *Proceedings of the USENIX Winter Technical Conference* (Dallas). 23–26.

Iamnitchi, A., Ripeanu, M., and Foster, I. 2004. Small-world file-sharing communities. In *INFOCOM '04* (Hong Kong). Vol. 2. 952–963.

Iyer, S., Rowstron, A., and Druschel, P. 2002. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing* (Monterey). 213–222.

Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. 2000. Oceanstore: An architecture for global-scale persistent storage. In *the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge MA). Vol. 28. 190–201.

Lee, E. and Thekkath, C. 1996. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge MA). 84–92.

Lee, J., Ma, X., Ross, R., Thakur, R., and Winslett, M. 2004. RFS: Efficient and flexible remote file access for MPI-IO. In *Proceedings of the IEEE International Conference on Cluster Computing* (San Diego). 71–81.

Lee, J., Ma, X., Winslett, M., and Yu, S. 2002. Active buffering plus compressed migration: An integrated solution to parallel simulations' data transport needs. In *Proceedings of the 16th ACM International Conference on Supercomputing* (New York). 156–166.

Litzkow, M., Livny, M., and Mutka, M. 1988. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems* (San Jose). 104–111.

Markatos, E. 2002. 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* (Berlin). 65–74.

Morris, J. H., M.Satyanarayanan, Conner, M., Howard, J., Rosenthal, D., and Smith, F. 1986. Andrew: A distributed personal computing environment. *Communications of the ACM 29,* 3, 184–201.

Muthitacharoen, A., Morris, R., Gil, T., and Chen, B. 2002. Ivy: A read/write peer-to-peer file system. In *Proceedings of the OSDI* (Boston). 31–44.

ncbi 2005. National center for biotechnology information. http://www.ncbi.nlm.nih.gov/.

Novaes, R., Roisenberg, P., Scheer, R., Northfleet, C., Jornada, J., and Cirne, W. 2005. Non-dedicated distributed environment: A solution for safe and continuous exploitation of idle cycles. *Scalable Computing: Practice and Experience 6,* 3, 107–115.

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

Otoo, E. J., Rotem, D., and Romosan, A. 2004. Optimal file-bundle caching algorithms for data-grids. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing* (Pittsburgh). 6.

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

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S. 2001. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM* (San Diego). 161–172.

Rivest, R. 1992. *The MD5 Message-Digest Algorithm.* Network Working Group RFC1321.

Sarkar, P. and Hartman, J. 1996. Efficient cooperative caching using hints. In *Proceedings of the 2nd ACM Symposium on Operating Systems Design and Implementation (OSDI)* (Seattle). 35–46.

Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C. 1990. Coda: A highly available file system for a distributed workstation environment. *ACM Transactions on Computer Systems 39,* 4, 447–459.

Schmuck, F. and Haskin, R. 2002. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of the First Conference on File and Storage Technologies* (Monterey). Number 19. 231–244.

SHARMAN NETWORKS, Inc. 2005. The kazaa media desktop. http://www.kazaa.com.

Strickland, J., Freeh, V., Ma, X., and Vazhkudai, S. 2005. Governor: Autonomic throttling for aggressive idle resource scavenging. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing* (Seattle). 64–75.

Szalay, A. and Gray, J. 2001. The world-wide telescope. *Science 293,* 14, 2037–2040.

The Astrophysical Research Consortium. 2005. Sloan digital sky survey project book. http://www.astro.princeton.edu/PBOOK/welcome.htm.

Thekkath, C., Mann, T., and Lee, E. 1997. Frangipani: A scalable distributed file system. In *Proceedings of the 16th Symposium on Operating Systems Principles* (Saint-Malo). Vol. 31. 224–237.

Vazhkudai, S. and Schopf, J. 2003. Using regression techniques to predict large data transfers. *High Performance Computing Applications - Special Issue on Grid Computing: Infrastructure and Applications 17,* 3, 249–268.