Positioning Dynamic Storage Caches for Transient Data

Sudharshan S. Vazhkudai* Douglas Thain† Xiaosong Ma‡ Vincent W. Freeh‡

Abstract

Simulations, experiments and observatories are generating a deluge of scientific data. Even more staggering is the ever growing application demand to process and assimilate these datasets. Application users perform a range of data operations, collaborate and share data in many novel ways. The current storage landscape is struggling to keep up with these trends in scientific data processing. Application users pay the price due to over-crowded shared filesystems, or expensive storage area networks, or not enough local storage, or high-latency archival or wide-area transfers.

In order to sustain and maximize I/O bandwidth relative to increasing CPU speeds, applications must take advantage of large amounts of intermediate commodity storage, However, intermediate storage presents new challenges above and beyond the traditional distributed filesystem paradigm: persistent scheduling, storage/CPU coallocation, namespace management, lifetime management, and novel application interfaces. In this paper, we describe applications that require intermediate storage management, suggest several open research problems, and illustrate two systems – Freeloader and Tactical Storage – that attack different aspects of these problems. [1]

1 Introduction

Scientific applications have become increasingly data-intensive [13]. State-of-the-art instruments and ultra-scale supercomputers generate terabytes of data per day. As a consequence, data storage and movement have become increasingly important and bottleneck-prone. Despite being equipped with layers of storage devices and application software, users often find the simple task of accessing data painfully difficult, labor-intensive, and slow.

We propose that the current storage hierarchy (Figure 1) available to scientists lacks the combination of capacity, performance, and life-span management to support accesses to large volumes of scientific data. Specifically, each layer of the scientific computing storage hierarchy caters to a subset of characteristics of scientific data production/consumption:

- **Parallel File Systems.** Large capacity and high performance is required to efficiently stage data from or to main memory and utilize the fast-growing aggregate compute power of today's parallel computers. Parallel file systems provide both capacity and performance, while also being widely used. However, due to the shared nature of most large machines and the data volumes, users are not allowed to hold their job input/output data indefinitely in a parallel file system scratch space. Typically, a purge policy requires users to orchestrate transfers of their data out of the scratch space within days of job completion. Therefore, parallel file systems are mostly used for one-time data generation or consumption of to be run or running jobs.

- **Mass Storage Systems.** Users hesitate to discard their data collected from instruments or simulations. Mass storage provides the large capacity and long-term storage space for safely archiving bulk datasets. Its obvious limitation is the performance hit due to latency: data has to be moved out for processing and tape access, combined with long-distance data transfer inhibits efficient direct stream processing. Since scientists may revisit the same datasets multiple times, during the days and weeks after the experiment/simulation that generated the data, they may have to move data back and forth between their local storage and a mass storage system, especially if they have very limited local storage space.

- **Distributed File Systems.** Ultimately scientists analyze and visualize data at their home institution, often far away from the experiment or computation site. File servers, accessed through distributed file systems, are

*Computer Science and Mathematics Division, Oak Ridge National Laboratory {vazhkudaiss}@ornl.gov

†Computer Science and Engineering Department, University of Notre Dame {dthain}@cse.nd.edu

‡Department of Computer Science, North Carolina State University {xma,vwfreeh}@ncsu.edu

[1]This work is supported in part by a DOE ECPI Award (DEFG02-05ER25685), an NSF CAREER Award (CNS-0546301), an IBM UPP award, a DOE contract with UT-Battelle, LLC (DEAC05-00OR2275), and Xiaosong Ma's joint appointment between NCSU and ORNL.

1-4244-0328-6/06/$20.00

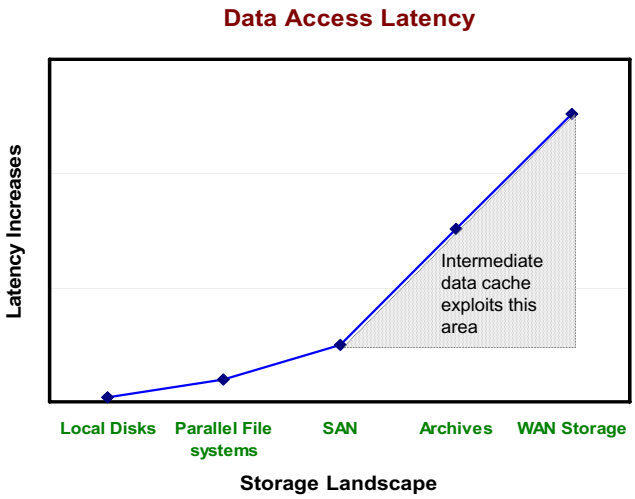

Figure 1. Caching in the Storage Landscape

The x-axis denotes current technology options in today's storage landscape. The y-axis denotes increasing latency. The figure plots a symbolic curve, depicting an increase in data access latency as we move from tightly-coupled to loosely-coupled storage. The figure denotes how intermediate storage caches can exploit the area under the curve for high-latency repositories.

available at research institutes. They provide shared file space, reasonable performance, and fault-tolerance through regular backups. These systems, however, are intended for ordinary files such as mails, documents, and web pages, which do not demand either large capacity or high data access performance. Moreover, users are often given limited quota on such systems, making them unsuitable for bulk scientific data.

- **Storage Appliances.** Users may choose to own dedicated storage servers. In exchange for the quota-free space availability, storage resources are often underutilized due to the bursty nature of scientific data processing (as opposed to, for example, business or web data hosted at typical storage servers). Plus, the cost of purchasing and subsequently managing private storage servers with the capacity to accommodate one's peak transient data needs often makes this solution out of reach for individual scientists or research groups.

In this paper, we recognize the above mentioned inadequecy of storage support for scientific data, and envision several solutions. In particular, we argue for the need for a layer of *intermediate storage or data caches* that provides longer-term storage than parallel file systems, larger capacity than desktops and distributed file systems, better access performance than mass storage, and smaller cost than dedicated storage servers. We discuss two novel approaches, namely FreeLoader [27] and Tactical Storage [25] that address the above issues in fundamentally different ways. Such intermediate storage would utilize the distributed and often existing storage fabric, and provide high performance to support transient accesses to large datasets.

For instance, as suggested by Figure 1, intermediate storage abstractions could reside anywhere between users and high-latency sources such as archival or other wide-area storage repositories. Depending on what constitutes their composition, they can offer varied benefits beyond the obvious "performance impedance matching". Intermediate storage caches built from aggregating commodity workstation storage such as FreeLoader offers low-cost storage at high-speeds due to parallel I/O. In addition, it enables locality of reference and cache behavior. Tactical Storage on the other hand, allows the easy construction of complex storage abstractions with relative ease without administrative support. It exports the full filesystem interface of each participating node into a larger file system for better space management and coarse-grained parallelism.

While there are significant advantages to using intermediary storage caches, their use in scientific computing has been gaining momentum only recently [8]. The world of Web proxy caching [2, 12, 9] and Internet content distribution [1] has long since exploited their use. In a similar vein, the Internet Backplace Protocol (IBP [18]) provides a low-level time limited allocation interface of storage extents across a distributed system.

Construction of intermediate storage caches is not without its challenges. First, given their potential use for transient data, the system should be able to adapt to dynamic access patterns and resource availability. Second, these storage systems should be able to support lifetime management

for datasets. Third, and in many cases, storage is required to be coscheduled with processing, which poses unique problems. Several other issues such as providing optimized I/O interfaces and virtualized access compound the problem.

In this paper we explore the design space of these intermediate storage caches; put forth and analyze several use cases in data intensive HPC settings (Section 2) that demand the need for sophisticated data management; derive a few key functionality (Section 3) desired from intermediate caches; and describe and compare two instances, namely FreeLoader and Tactical Storage (Section 4).

2 Use Cases for Intermediate Storage

In this section we put forth several scenarios and use cases that call for the need for sophisticated data caches and alternate data management approaches than what is available in the main stream currently.

Client-side caching: The use of client-side caching as a means of expediting data access is well known (i.e. web proxy caches and content distribution networks.) The scientific computing world, however, lacks similar support for powerful caching that utilizes distributed storage resources to transparently accelerate accesses to remote data repositories. Yet the need for such caching is becoming more evident and urgent. First, local data processing is becoming more prevalent due to ever increasing desktop capabilities and convenience in terms of operating environments and visualization tools. Second, repeated wide-area downloads is often not practical since they are stymied by several bandwidth/latency issues and have to be specifically tuned for optimal rates. As Gordon Bell, Jim Gray and Alex Szalay put it in their recent article on Petascale Computational Systems, "it is not worthwhile to move large amounts of data unless performing analysis, requiring more than 100,000 CPU cycles per byte of data" [5]. Third, large datasets 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. 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. However, beyond this processing duration, users normally choose not to retain copies of the downloaded datasets locally. This suggests that local-area, client-side caches can be very useful in hiding latency, promoting data reuse by exploiting temporal locality and in providing more storage than what is available for a desktop user. Further, one can imagine building more sophisticated partial caching techiniques atop such client-side caches that only hold a prefix of the dataset, while the suffix is patched transparently.

Storage caches and checkpointing: Several high-end applications run on thousands of processors, are high-throughput, long running and checkpoint terabytes of data as snapshots for recovery in the event of failure or system crash. Checkpoint data is usually written once and "hope to be never used" unless a failure occurs. Checkpointing terabytes of data to conventional parallel file systems with strict I/O semantics can be very time consuming. To applications, this is time spent away from useful computation.

Meanwhile, existing reliability, availability, and serviceability (RAS) approaches have not been able to fully exploit opportunities in the high-end system I/O stack, ranging from applications I/O semantics to available hardware/data redundancy. Overly conservative file system consistency fails to efficiently overlap I/O with computation and forces unnecessary inter-processor synchronization, while the local disk storage or memory attached to individual nodes on supercomputers and clusters has been largely ignored in the I/O stack. For example, checkpoint data has varied persistence requirements and access patterns, and can benefit greatly from being stored at a peer-processors local disk/memory, or a cache using an architecture that makes them available through a relaxed, convenient I/O interface. This offers enhanced scalability while reducing access latency. Effectively utilizing these resources and opportunities will bring significant improvements to the availability and performance of the I/O system in large-scale machines.

Localized Intermediate Results: In many scientific fields, it is common to generate large amounts of data through simulation, but only retain a selection of results that are of sufficiently high quality. For example, when studying molecular dynamics, one may use monte carlo methods to generate an array of proposed transition paths between conformations, but then only retain the lowest energy transition. Such outputs are similar to checkpoints in the sense that they need not be organized for maximum persistence: if lost, the outputs can simply be generated again. However, they do have some aspects of permanent storage: they must still be named and kept in a manner that later computations (codes that rank the "quality" of a result) can locate, access, and possibly extract data for a limited time after creation.

To support localized intermediate results, a storage system must provide a large amount of space close to processing units. It need not have the throughput of a parallel file system, nor the reliability of an archival storage system, but it must have the capacity of scratch space and the easy addressability of a distributed file system.

Staging, Offloading and Failovers: Modern supercomputer centers will have thousands of user jobs waiting to be run, each with several orders of gigabytes or terabytes of input data. These input data have to be either staged on general-purpose parallel file systems for ready availability once the job is scheduled or pulled from wide-area resources or archival systems. In the former case, high-end resources (such as storage and processors) are almost al-

ways busy with long queues of pending requests; in the latter, the job is wasting invaluable allocation time, performing expensive I/O operations. Similar reasoning holds for post processing data, which is required to be pulled quickly out of the compute nodes. Due to end-resource unavailability, data migrations can be temporarily staged in intermediate storage along the data pathway. The upshot is that, using conventional parallel file systems attached to supercomputers for staging input/output or checkpoint data (which might be seldom used) may not always be a suitable option for supercomputing facilities. This calls for new techniques through the use of storage caches.

Further, data unavailability in the underlying file systems can often translate into expensive reconstruction from archival storage or remote source copies, which incurs high latency. A gigabyte dataset accessed from HPSS tape archives [7] delivers a rate of 10MB/sec despite end resources being connected through gigabit networks [17]. Such high latency is unacceptable to many applications. In such cases, data failover to nearby storage caches that complement general purpose file systems can be a very useful technique masking failure.

Datacenter caches: Parallel file systems attached at computing facilities are, in most cases, not intended to be used as the long-term storage repositories for scientific datasets. User data generated from a computation job is typically purged from the scratch space within a short period of time after the job is completed (typically hours to days). Scientists' common practice is to move the data to mass storage systems, or their local data processing clusters, or both. There is a current trend, however, for scientific communities or an experimental facility to build shared data centers. These data centers consist both mass storage and parallel computing facilities that provide integrated data storage, management, query, and processing services. Examples of such data centers include the Sloan Digital Sky Survey (SDSS [20]), the Spallation Neutron Source (SNS [6]) and Earth System Grid (ESG [3]), which allow scientists to obtain high-performance interactive processing of remotely hosted data with little or no hardware investment and to avoid expensive wide-area data transfers.

With such data centers, where the bulk of data is kept on tape archiving systems, the gap between the processor speed and the I/O rates is even more dramatic. The I/O performance is especially a concern when such data centers support online data queries and visualization tasks through data gateways or portals (e.g., SNS, ESG). To bridge this performance gap, disk storage equipped at the data center's computing facility must be carefully used to cache hot datasets and reduce tape I/O as much as possible. However, achieving this requires automated tools, rich caching and data aging techniques to dynamically stage data based on portal-user data access patterns. SRM [21] addresses some of these issues with caching. Such gateway data caches allows facilities to optimally manage online storage while also hiding the latency involved in accessing archival storage. In addition, they allow for the data management infrastructure to stage an active window of datasets, comprising of most recently used user data.

Aforementioned are a few data intensive operations in HPC settings that require a novel data management layer for efficient processing.

3 Desired Functionality of Caches

Several of the above usecases suggest the following desired functionality in intermediate storage systems.

Scalability of Performance and Capacity: Distributed storage caches must be prepared to deal with the unlimited I/O appetite of scientific appcliations as well as interact with high performance parallel file systems and large capacity of archival storage systems. Because such caches will typically be constructed from commodity hardware, the software structure must permit expansion of both throughput and capacity by the incremental addition of new nodes.

Ability to reserve and allocate space: Users need to reserve and allocate space in the storage system so they can upload or download data in the future. The ability to reserve space has a two-fold benefit. First, it allows users to perform advance planning and scheduling for the concerted use of available resources. Second, it helps the storage system to confirm user intent to use the allocated space. It is thus desirable for a storage system to provide such capabilities to handle dynamic user access patterns.

Transient data with temporal validity: A common thread in many of the usecases put forth earlier is the need to manage transient data. Increasingly, users wish to download and analyze large datasets for limited durations of time. Or, data is required to be staged in a storage system at a computer center until it is pulled for processing. Or, checkpoint data is required to be maintained until job completion. This suggests that associating temporal validity to datasets and providing related mechanisms to operate on them can be very useful for the underlying storage system to offer.

Relaxed POSIX I/O semantics: Data intensive I/O in HPC settings manipulate terabytes of data and include operations such as staging, checkpointing, prefetching, etc. In many cases, they do not require strict POSIX semantics, but often have to deal with the legacy of file system implementations. For instance, checkpoint operations are write once and hope to be never read. However, applications often checkpoint terabytes of data for recovery purposes. Can the performance of several such operations be improved using relaxed POSIX or optimized read/write semantics?

Application-specific optimizations: Exploiting commonly observed data access patterns in HPC settings and

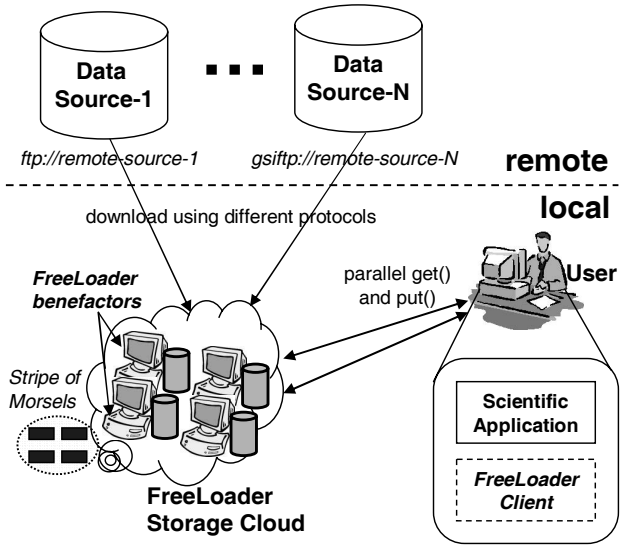

Figure 2. FreeLoader Environment

optimizing storage system behavior can be beneficial. For instance, several data intensive applications perform large, sequential I/O reads; can the storage system exploit such scientific data properties of applications for better performance?

Virtualizing storage access: User applications often desire seamless, transparent mechanisms to access disparate storage entities available to them so I/O accesses can progress smoothly. For instance, automatically accessing caches or archives from file systems can be a useful prefetching or recovery strategy. Thus, virtualizing access to these storage resources is highly desired in HPC settings.

Easy-to-use interfaces: Aforementioned features are some of those desired in storage systems for data intensive applications. However, if they are not packaged as easy-to-use interfaces that users/applications are familiar with, they are not of much use. For instance, applications can remain oblivious to how underlying storage systems are constructed (distributed or centralized; commodity or dedicated) or store datasets (single file objects or striped) as long as they can be accessed using regular file I/O mechanisms. Further, it is essential for storage systems to be wide-area accessible, and hence agnostic to various protocols. Thus, it is essential to provide regular I/O interfaces. Also, with the proliferation of MPI-IO [11, 14, 16, 19, 4] for scientific applications, it would be beneficial for new storage abstractions to interface to the same.

4 Case Studies in Intermediate Storage

In this section, we give an overview of two systems – Freeloader and Tactical Storage – that address some of the challenges we have outlined for intermediate storage.

4.1 FreeLoader: Desktop Storage Caching

FreeLoader [27] is a distributed storage framework, (Figure 2), that provides abundant, high-performance site-local storage for scientific datasets with very little additional expense, by aggregating idle desktop storage resources. It is a near-the-client network data cache that aggregates donated storage from collaborating user workstations into a single storage cache. FreeLoader aims to aggregate both distributed storage resources and I/O as well as network bandwidth. Workstation owners within a local area network donate unused disk space, and FreeLoader stripes datasets onto multiple such workstations to enhance data access rates. It stores large, immutable datasets by fragmenting them into smaller, equal-sized chunks (morsels), which are scattered among the storage nodes. This enables each researcher in the group to process the raw datasets as if they reside on a high-performance shared file system, increasing collaboration and reducing expensive downloading operations.

Within each FreeLoader instance, a dedicated manager node maintains metadata such as node status, chunk distribution, and dataset attributes including the primary copy location (URI and, if necessary, authentication related metadata). A participating workstation may be a storage node that donates disk space along with I/O and network bandwidth, or a client node that stores/retrieves data from the FreeLoader space, or both. Data storage and retrieval are initiated by the client via the manager, while the actual transfer of data chunks occur directly between the storage

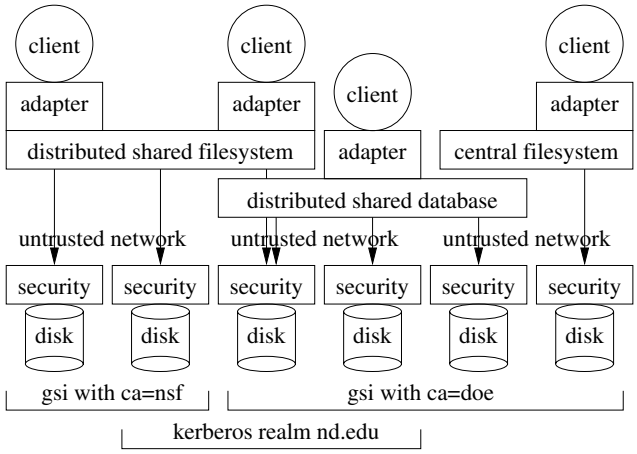

Figure 3. Overview of Tactical Storage

A tactical storage system is composed of a collection of low level file servers that export a secure filesystem interface. One or more servers can be joined together into higher level abstractions and joined to legacy applications via user-level adapters.

nodes and the client.

FreeLoader is a storage system and not a filesystem. However, it provides I/O interfaces such as open, read, write and close for applications to access data from the storage cloud. The read and write calls are translated into FreeLoader chunk transfer operations, with additional processing such as data trimming and concatenation. FreeLoader I/O is optimized for reading large, sequential datasets and achieves upto a three-fold increase in throughput compared to several storage systems. With this basic infrastructure and relaxed I/O interfaces, FreeLoader can be used as a client-side cache to which users can read and write large datasets using standard file interfaces.

FreeLoader treats the entire aggregated space as a cache by performing cache replacement and evictions to create room for new, in-coming datasets. Cache eviction is at the granularity of datasets using LRU and within datasets, at the granularity of chunks. Since FreeLoader is optimized for sequential accesses, chunks within a dataset are evicted from the tail. Therefore, if only part of a dataset needs to be evicted, a prefix of that dataset will still remain in the cache for subsequent accesses.

FreeLoader also provides sophisticated prefix caching strategies [17]. By overlapping in-cache data access with data retrieval directly from the primary copies, prefix caching helps in maximizing space utilization and increasing cache hit ratio by offering a virtual cache that appears to be larger than the physically available cache space. Compared to caching entire datasets, users enjoy higher hit rate without suffering degraded data retrieval performance while accessing cached datasets. To this end, FreeLoader has built-in techniques to fetch partial pieces of data seamlessly from remote storage repositories, thus providing virtualized storage access. With such rich caching semantics, FreeLoader can be used for transient data with certain temporal validity. This is a particularly useful feature when it comes to being used as datacenter caches or for staging and offloading datasets in a supercomputer center.

With its distributed nature, parallel I/O across a network of workstations, application-specific optimizations and caching, FreeLoader is strategically poised for use for a variety of HPC scenarios.

4.2 The Tactical Storage System

The Tactical Storage System (TSS) [25] is designed to maximize the ability of the end user to construct and configure complex distributed storage systems without requiring administrator intervention. The key property of a TSS is that it separates control over *resources* (the raw storage and networking capacity) from control over *abstractions* (the organization of resources). This permits end users to construct and access complex storage structures over the wide area without requiring an interaction with the administrator each time a new configuration or activity is needed. We call this *tactical* because it permits the user to harness storage in arbitrary locations for the short-lived needs of remote computation.

Figure 3 shows the primary structures of a TSS. The foundation is an array of user-level file servers that may be deployed on any collection of conventional machines. These machines need not have any specialized roles. For example, at the University of Notre Dame, a standing collection of 200 file servers is deployed across a wide array of personal workstations, classroom labs, and computing clusters. Each server exports a Unix-like I/O interface with a flexible security interface, allowing for the owner of the

storage to control who may access the storage, and how much space and bandwidth they may consume. More recent work allows external users to request guaranteed space allocations [24] within the space allotted to a file server and to execute remote code in a secure *identity box* [23] in the manner of active storage.

The security mechanism of a TSS allows users to be authenticated by a variety of techniques including simple hostnames, Kerberos [22], and the Grid Security Infrastructure (GSI) [10]. Authorization is achieved through per-directory ACLs in a manner similar to that of AFS. The appropriate mechanism may be chosen at run-time based on the needs of the application: a given server may allow access to a few trusted hosts based solely on network address, while also allowing access to remote users with stronger credentials. In addition, distributed ACLs allow access controls on one file server to refer to groups defined on another [15].

Above these file servers, end users may construct a variety of *abstractions*. Each abstraction is implemented using the available file server interface, and requires no particular cooperation with the storage provider beyond conformance to the local security policy. The simplest abstraction is a *central file system* (CFS) which gives an external user a direct view of a remote filesystem. This is roughly analogous to providing an implementation of secure wide-area NFS. Multiple file servers may be tied into a *distributed shared filesystem* (DSFS), which allows for the construction of arbitrarily large filesystems. One file server is used as a central directory server, which contains pointers to files scattered across the remaining disks. Multiple file servers may also be tied into a *distributed shared database* (DSDB), which relies on a central database server to index the locations and properties of file objects scattered around the remaining disks.

Storage abstractions are of little use unless applications can actually connect to them. To this end, the TSS provides an *adapter* called Parrot [26] that presents each storage abstraction as a conventional filesystem. The adapter runs ordinary applications and uses the system debugging interface to capture and interpret system calls. I/O system calls that refer to the TSS are captured and implemented on behalf of the application. The important property of this technique is that it can be applied entirely at user level and requires no privileges or kernel changes even to install, thus it can be used even by applications running in remote batch systems. To the end user, the TSS looks and acts just like a kernel-implemented distributed filesystem.

Finally, the adapter allows for the construction of *custom name spaces* for individual applications. While individual abstractions can be named and accessed explicitly through the filesystem (`/cfs/host.nd.edu/mydata`), we assume that applications will tend to access data through fixed logical path names. To this end, the adapter provides a *mountlist* facility that maps logical names to physical addresses, for example:

```
/data        /cfs/host.nd.edu/mydata
/usr/local   /dsfs/archive.nd.edu/sw
```

In a typical use case, the TSS would be used as a facility for *explicitly* staging data in and out of a cluster of commodity storage devices. For example, to bring a large dataset close to a batch computing facility, the user could create a DSFS abstraction spread across eight disks, copy the data from a local filesystem into the cluster using ordinary `cp` commands, and then execute jobs in the batch system, referring to the data staged into the cluster. Using the transparent adaptation facility, data in the cluster can be seen and manipulated like a local filesystem from anywhere on the network. When complete, needed data can be copied out of the DSFS and then the entire abstraction deleted with a single command.

4.3 Comparison

A brief comparison of Freeloader and TSS is instructive to understand better what problems each system addresses and to what applications each is suited. We pick a few key areas such as parallelism, space management, data transience and namespace to illustrate how the two systems differ from each other. The intent is not to suggest the use of one or the other. On the contrary, the two systems can coexist as we will show. However, it is intended to illustrate the spectrum of possibilities intermediate storage systems can choose to implement in several of these key areas.

Parallelism. Freeloader is designed to extract high I/O bandwidth from commodity storage clusters through a high degree of parallelism. It is meant to serve as a cache for or even replacement for a high end storage archive that delivers constant high throughput to a single application. To this end, large storage objects are broken up into multiple pieces and spread across multiple devices. A single application may achieve high I/O bandwidth by requesting multiple blocks in parallel.

Tactical Storage is designed to make many instances of local storage accessible to a wide variety of users and applications, particularly those that run in distributed batch systems. To maximize the utility of any single node, each exports a full filesystem interface that can be used independently. Although multiple file servers can (and are) aggregated into larger filesystems and databases, this facility is used to increase the overall *throughput* of the system to process sequential jobs on partitioned data.

In summary, Freeloader provides fine-grained parallelism in order to maximize I/O bandwidth for a small number of large applications, while Tactical Storage provides

coarse-grained parallelism in order to maximize throughput for a large number of small applications.

Space Management. Freeloader harnesses the unused storage space found on personal workstations while attempting to minimize the impact of this consumption on the local user. If a local user signals an intent to consume more space, then data must be deleted from that node. As a result, the broader system must incorporate a certain degree of data replication and fault tolerance to compensate.

Although Tactical Storage can also run on a collection of workstations, it assumes that each workstation gives a fixed allocation of space to be used by each storage server. This allocation may then be further subdivided and issued to remote users. Clearly, this arrangement does not maximize the total storage, but it does provide a certain degree of guarantee to applications: if the allocation succeeds, an application can be relatively sure of success.

In summary, Freeloader scavenges for unused space, while Tactical Storage exports fixed allocations of space. The best choice depends on the dynamics of the system in question.

Data Transience. Freeloader can function as a buffer cache for remote storage archives. As applications refer to data items in Freeloader, they are paged in and cached on demand. In addition, more recent work on prefix caching [17] allows for the user to explicitly trigger the loading of a dataset into a Freeloader cache asynchronously of the application's actual access to the cache.

Tactical Storage encourages the use of file servers as staging points in a distributed computation. The same file server and protocol can be used for both streaming data movement as well as filesystem access, allowing the user to easily push data to a remote site, and then access it transparently. Using the namespace function in the adapter, the application will not perceive any external difference. However, the user (or the broader system) must explicitly arrange the transfer and configuration.

In summary, both Freeloader and Tactical Storage encourage the use of intermediate storage for transient data. However, Freeloader provides a transparent buffer cache, while Tactical Storage encourages explicit data staging.

Application Namespace. FreeLoader is a storage cache and not a file system. However, it is vital to provide a namespace for application accesses. Current Freeloader implementation supports a flat namespace wherein datasets are identified using the URI with which they are imported into the storage cloud. For instance, if a dataset was imported using the HTTP protocol from "from-this-url/dataset", then it is identified by the URI, "http://from-this-url/dataset". This allows FreeLoader to perform easy recovery from remote sources.

Tactical Storage attempts to provide a namespace as close as possible to the filesystem directory hierarchy expected by most applications. Remote file services are seen as filesystem entries containing the name of the hosting machine. In addition, the namespace mechanism allows for the mapping of logical directories to physical locations.

To summarize, these two systems complement each other. Tactical Storage can be used to provide a filesystem abstraction atop FreeLoader so applications can use FreeLoader capabilities using familiar interfaces. Given the range of different requirements these two systems address, such an endeavor can be extremely useful in supporting a variety of data intensive HPC applications.

5 Future Work

Combining Freeloader and Tactical Storage. Although FreeLoader and Tactical Storage address different aspects of storage caches – Freeloader maximizes bandwidth for single clients, while Tactical Storage maximizes throughput for many jobs – there are opportunities to combine the technologies. TSS provides a transparent adapter that connects Unix applicatinos to remote storage. This adapter could be modified to address and load data from Freeloader, making it appear as a conventional file system. Conversely, TSS provides low-level storage devices that Freeloader could harness as storage benefactors, thus increasing the set of resources available for caching. We plan to address both of these directions in future work.

Seamless Data Migration. As we discussed earlier in this paper, the storage landscape includes a wide variety of existing systems: parallel, archival, distributed, appliance, and now storage caches. Users would be much better served if data was easily migratable between each of these kinds of systems. In the same manner that hierarchical storage systems migrate data between disk and tape, users need to be able to migrate data between each category of storage system, depending on the data processing needs of the moment.

Such migration is currently difficult because each category of system already has several distinct software infrastructures. (In this paper alone, we have introduced two distinct infrastructures for storage caches.) We cannot expect any single software infrastructure to be deployed on all of these system categories. Yet, there is an opportunity for a higher layer of software to unify multiple disparate storage systems, allowing for transparent access to logically-named data, regardless of the category of system it resides in.

Performance Adaptation. As we have pointed out, each category of system offers a different tradeoff between I/O bandwidth, access latency, lease time, and monetary expense. Given a framework for seamless data migration between categories of machines, it should be possible to construct a system that migrates data between categories in the storage hierarchy, based on the runtime behavior (and advance requests) of clients. In the same way that a conven-

tional operating system provides a memory based buffer for slow disks, a storage manager should be able to transparently migrate and reconfigure data between caches based on client needs.

6 Conclusion

In this paper, we present the need for novel intermediate storage nodes or caches as a means to address the gap in the current storage landscape for data intensive HPC applications. We explore the design space of storage caches and present several data intensive scenarios as usecases. Intermediate storage can be used in a variety of cases such as client-side caching, staging, offloading, checkpointing, etc. The various scenarios also illustrate certain desired behavior from these systems. These include functionalities such as lifespan management of datasets, relaxed, fast, easy to use I/O mechanisms, ability to allocate space, etc. We then present two case studies, namely FreeLoader and Tactical Storage, as major implementation examples that address different requirements.

References

[1] Akamai. http://www.akamai.com/, 2005.

[2] Squid web proxy cache. http://www.squid-cache.org/, 2005.

[3] Earth system grid. http://www.earthsystemgrid.org, 2006.

[4] T. Baer and P. Wyckoff. A parallel i/o mechanism for distributed systems. In *Proceedings of the International Conference on Cluster Computing*, 2004.

[5] G. Bell, J. Gray, and A. Szalay. Petascale Computational Systems. *IEEE Computer*, 39(1):110–112, 2006.

[6] J. Cobb, G. Geist, J. Kohl, S. Miller, P. Peterson, G. Pike, M. Reuter, T. Swain, S. Vazhkudai, and N. Vijayakumar. The neutron science teragrid gateway: a gateway to support the spallation neutron source. *Software Concurrency and Practice*, 2006.

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

[8] Data capacitor. http://datacapacitor.org.

[9] B. Davison. Web caching and content delivery resources. http://www.web-caching.com/, 2005.

[10] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computer and Communications Security Conference*, 1998.

[11] T. Fuerle, E. Schikuta, C. Loeffelhardt, K. Stockinger, and H. Wanek. On the implementation of a portable, client-server based MPI-IO Interface. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 5th European PVM/MPI Users' Group Meeting*, 1998.

[12] S. Gadde, J. Chase, and M. Rabinovich. A taste of crispy squid. In *Proceedings of the Workshop on Internet Server Performance*, June 1998.

[13] J. Gray and A. Szalay. Scientific data federation. In I. Foster and C. Kesselman, editors, *The Grid 2: Blueprint for a New Computing Infrastructure*, 2003.

[14] R. Hedges, T. Jones, J. May, and R. Yates. Performance of an MPI-IO implementation using third-party transfer. In *Proceedings of the 17th IEEE Symposium on Mass Storage Systems*, 2000.

[15] J. Hemmes and D. Thain. Cacheable decentralized groups for grid resource access control. In *IEEE Grid Computing*, September 2006.

[16] J. Ilroy, C. Randriamaro, and G. Utard. Improving MPI-I/O performance on PVFS. In *Proceedings of the 7th International Euro-Par Conference*, 2001.

[17] X. Ma, S. Vazhkudai, V. Freeh, T. Simon, T. Yang, and S. Scott. Coupling Prefix Caching and Collective Downloads for Remote Dataset Access. In *Proceedings of the 16th ACM International Conference on Supercomputing*, 2006.

[18] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Proceedings of the Network Storage Symposium*, 1999.

[19] J. Prost, R. Treumann, R. Hedges, A. Koniges, and A. White. Towards a high-performance implementation of MPI-IO on top of GPFS. In *Proceedings of the 6th International Euro-Par Conference*, 2000.

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

[21] A. Shoshani, A. Sim, and J. Gu. Storage resource managers: Essential components for the grid. In J. Nabrzyski, J. Schopf, and J. Weglarz, editors, *Grid Resource Management: State of the Art and Future Trends*, 2003.

[22] J. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Winter Technical Conference*, pages 191–200, 1988.

[23] D. Thain. Identity boxing: A new technique for consistent global identity. In *International Conference for High Performance Computing, Networking, and Storage (Supercomputing)*, November 2005.

[24] D. Thain. Implementation tradeoffs in storage allocation for grid computing. In *IEEE Grid Computing*, September 2006.

[25] D. Thain, S. Klous, J. Wozniak, P. Brenner, A. Striegel, and J. Izaguirre. Separating abstractions from resources in a tactical storage system. In *International Conference for High Performance Computing, Networking, and Storage (Supercomputing)*, November 2005.

[26] D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. In *Workshop on Adaptive Grid Middleware*, New Orleans, September 2003.

[27] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. Scott. Freeloader: Scavenging desktop storage resources for bulk, transient data. In *Proceedings of Supercomputing*, 2005.