Coupling Prefix Caching and Collective Downloads for Remote Dataset Access

Xiaosong Ma
Vincent W. Freeh, Tao Yang
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206
{xma,vwfreeh,tyang2}@ncsu.edu

Sudharshan S. Vazhkudai
Tyler A. Simon, Stephen L. Scott
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831
{vazhkudaiss,simonta,scottsl}@ornl.gov

Abstract

Scientific datasets are typically archived at mass storage systems or data centers close to supercomputers/instruments. End-users of these datasets, however, usually perform parts of their workflows at their local computers. In such cases, client-side caching can offer significant gains by reducing the cost of wide-area data movement.

Scientific data caches, however, traditionally cache entire datasets, which may not be necessary. In this paper, we propose a novel combination of prefix caching *and* collective download*. Prefix caching allows the bootstrapping of dataset downloads by caching only a prefix of the dataset, while collective download facilitates efficient parallel patching of the missing suffix from an external data source. To estimate the optimal prefix size, we further present an analytical model that considers both the initial download overhead and the downloading speed. We implemented our proposed approach in the FreeLoader distributed cache prototype. Experimental results (using multiple scientific data repositories and data transfer tools, as well as a real-world scientific dataset access trace) demonstrate that prefix caching and collective download can be implemented efficiently, our model can select an appropriate prefix size, and the cache hit rate can be improved significantly without hurting the local access rate of cached datasets.*

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; C.2.4 [**Compute-Communication Networks**]: Distributed Systems; C.4 [**Performance of Systems**]

ICS06 , June 28-30, Cairns, Queensland, Australia

1. Introduction

The use of client-side caching as a means to expedite data access is well known [16]. HTTP proxy caches have long been used for Web-based document browsing (Squid [2], etc.). Content delivery systems such as Akamai [1] extend this by way of deploying numerous surrogate, edge-cache servers closer to end-users that host popular Internet content. In general, client-side caching reduces servers' bandwidth consumption and load, but more importantly, reduces latency and increases client perceived throughput. These benefits are even more evident for large media files in HTTP downloads. In fact, it is reported that much of the usefulness of Web proxy servers—up to 80% for some installations—is in exploiting the locality involved in cached objects [16]. To this end, many proxy caches are part of cooperative cache hierarchies, wherein neighboring caches inquire each other to serve client requests (e.g., Squirrel cooperative desktop browser cache [22]).

In the scientific computing world, however, there 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:

- Generally, scientific computing workflows are distributed. Raw data is often generated/collected from shared supercomputers or experimental/observational instruments, while end-users of these data perform analysis and/or visualization at their home institutes. Expensive wide-area data movement is necessary in such cases.

- End-users are not able to store all their data locally. This is partly due to the ever increasing data volume in scientific computing [20, 19]. Further, scientists prefer to archive their datasets at shared repositories, such as the mass storage systems (e.g., HPSS [15]) and data centers (e.g., SDSS [38]). These systems often reside close to high-end computing or instrument facilities, with convenient and fast data transfer capabilities to and from the latter. They provide large capacity, fault tolerance, and easy data sharing among collaborators.

- There exists a performance "impedance mismatch" between the rates at which end-user applications consume data locally and retrieve data from shared repositories. Despite the

performance improvement in recent years, wide-area data transfers are the most common bottleneck in an end-to-end scientific data processing workflow [44]. Besides, even if available, high-speed transfer tools are required to be tuned to obtain optimal throughput and such tuning is not commonplace yet [44]. Because direct streaming of remotely archived data is unable to support the rate needed by data processing applications—especially interactive applications such as visualization—users typically move their datasets manually to local secondary storage before performing their data analysis.

- There is access locality in scientists' data analysis. Scientists often focus on newly retrieved datasets for a certain period of time, typically days or weeks. In their analyses they are also required to compare different batches of data to study the impact of varied computation models, parameters, or input. Limited by their local storage capacity, many scientists end up manually swapping data from the archiving repository, suffering the large latency and low bandwidth multiple times.

Motivated by the above facts, several near-the-client caches for scientific datasets have been put forth recently. These include dedicated deployments of large storage servers (IBP [35]), dedicated deployment of distributed parallel storage servers (DPSS [45]), and loosely-coupled, highly dynamic collaborative desktop storage caches (FreeLoader [47]). These caches enable faster data accesses by storing "hot" datasets close to their users.

However, existing systems cache scientific datasets in their entirety. In this paper, we show that doing so is not necessary, and propose a novel combination of *prefix caching* and *collective download*, two techniques originating from the multimedia data streaming and parallel I/O fields respectively. Prefix caching allows the storage of only partial datasets, while collective download allows seamless and fast parallel downloads of the uncached suffix. In particular, our proposed approach achieves efficient parallel data retrieval from external scientific data repositories by issuing large, sequential file transfer requests. It further maintains high local cache access performance by rearranging the data for finer-grained data striping. The upshot is an improved cache hit rate, without compromising the user-perceived access rate of cached datasets. Further, we demonstrate that given a dataset and its external data sources, we can effectively predict the appropriate prefix size to minimize its cache space consumption.

We study the above techniques within the context of the FreeLoader distributed storage prototype [47], which aggregates unused desktop disk space into a cache/scratch space for scientific datasets. In addition to performance benefits, the proposed combination of prefix caching and collective download also enables new resource aggregation models. Our experiments using the FreeLoader prototype and several scientific data repositories illustrate that collective download can be implemented efficiently to enable aggressive prefix caching. In addition, our trace-driven simulation using a real-world scientific dataset access log reveals that with the performance of today's data repositories, prefix caching can significantly improve the caching performance.

The rest of the paper is organized as follows. Section 2 gives a brief introduction on the FreeLoader system, and surveys related work. Section 3 presents our proposed approach and Section 4 discusses its implementation. Section 5 discusses performance results and Section 6 summarizes the contributions of the paper.

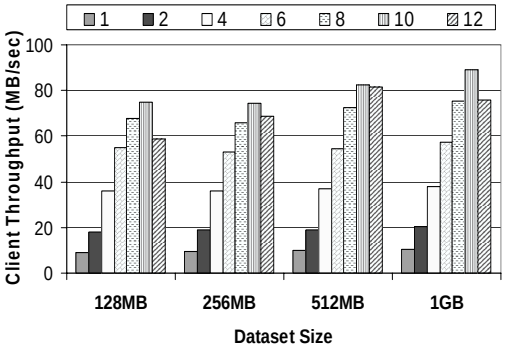

Figure 1. FreeLoader's client dataset retrieval throughput with different stripe widths.

2. Background

2.1 The FreeLoader Storage System Prototype

FreeLoader [47] 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, 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). Such metadata enable transparent suffix patching of datasets from their primary copies. 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 nodes and the client.

Figure 1 demonstrates FreeLoader's data access performance by showing the client-side dataset retrieval rate using different stripe widths and dataset sizes. The experiments were performed on a client workstation with a Gbit/s network interface and a group of storage nodes with mixed interfaces (Gbit/s or 100Mbit/s). The client's Gbit/s network configuration is meant to capture scientists' high-end desktop workstations with good connectivity for scientific data processing. Figure 1 shows that in such an environment, storage space aggregation through striping brings not only *increased capacity* by utilizing other workstations' unused disk space, but also *increased performance* by aggregating I/O

and network bandwidth at the distributed machines. Even when users can accommodate datasets at their own workstations, storing such data in an aggregated cache like FreeLoader will deliver an I/O throughput (as high as 88MB/s with a stripe width of 10) significantly better than local disk I/O throughput (typically 30-50 MB/s). Finally, results reported earlier [47] revealed that striping is also an effective way to reduce FreeLoader's performance impact on space donors' native workloads.

2.2 Related Work

Prefix caching [21, 40] techniques have been proposed and implemented for multimedia streaming protocols such as the IETF Real-Time Streaming Protocol [37] that is built on HTTP. Prefix caching is well suited for Web multimedia objects since it reduces the latency, startup delay, and jitter involved in streaming media files. However, if the streaming protocol itself is not inherently parallel, HTTP prefix caching cannot exploit the residual bandwidth that might be available between the proxy cache and the server. Also, HTTP prefix caching—or for that matter, HTTP caching in general—needs to address consistency issues that are endemic to Web data objects [40]. Our work applies prefix caching to scientific data storage/access, and leverages the parallel transfer capability offered by tools such as GridFTP [8, 9], HSI [18], and LoRS [34].

Middleman cache for video files on the Web is similar to our effort [6]. It is a cooperative proxy cache built from client workstations. However, it only exploits collaborating workstations for aggregate space and not for improved bandwidth. Another similar project on Internet streaming uses a combination of network bandwidth estimation between the client, cache and server to optimize media delivery [23]. We use similar information to determine the ideal prefix size.

Scientific data caches like IBP [35], DPSS [45], and HPSS-disk caches [15] all provide techniques to accelerate data accesses by offering dataset caching. However, they support only entire dataset caching and perform cache management by enforcing user quotas. Moreover, users explicitly have to create space for new incoming data by deleting datasets that they no longer require. In brief, cache replacement is not dynamic and is left to user discretion. HPSS disk caches is an exception to this since the cache—a collection of disks atop high-latency tapes—is a staging area for most recently accessed datasets on a per-user basis. Our proposed approach can potentially be used by the above systems for better cache space utilization, especially when parallel data transfer is available in retrieving uncached data.

Also related is middleware such as Storage Resource Manager, SRM (e.g., Storage Resource Manager, SRM [32, 41]). SRM provides an interface between caches and mass storage systems by queuing requests, negotiating transfers and handling failure, etc. Several science communities (e.g., climate [3, 4] and high energy physics) provide access to archival data to a distributed community using such systems (e.g., Earth System Grid, ESG [4]). However, SRM does not perform partial caching and is itself not a cache.

To a certain extent, our work is similar to BitTorrent-style peer-to-peer content distribution systems [10], in the sense that a node may be serving data while retrieving missing data from others. However, our approach targets a much less dynamic system, with predictable data flows (remote data repositories to local cache to clients), centralized control over the cache content, and tightly coupled parallel data transfer.

Our collective download approach resembles the collective I/O technique extensively studied in the parallel I/O field and widely used in parallel simulations [11, 25, 31, 39, 43]. Collective I/O attacks the I/O performance problem caused by a mismatch of data distribution in memory and in files by consolidating small, scattered I/O requests into large, sequential ones. Although collective I/O has been used in conjunction with wide-area data migration [27, 26], it was applied to the local staging step only. We extend this approach to parallel download, by issuing a small number of large, sequential partial file transfer requests, in order to achieve better overall downloading performance.

Note that the term "parallel download" often means downloading in parallel from *multiple* remote source copies. There have been previous studies on this subject in the context of co-allocated download for GridFTP [46], IBP [34], or regular Internet downloads [13, 36]. In this paper, "parallel download" means multiple nodes each using a separate stream to download parts of a file from a common source copy simultaneously. These two techniques are orthogonal and can potentially work together. One recent study particularly relevant to our work investigated optimizing parallel large file transfers between two sites where each site deploys data striping [48]. In contrast, our approach assumes no knowledge on the data distribution at the data sources (servers).

Finally, predictive prefetching is a widely researched topic to improve WWW as well as cache performance [33, 24]. This can also be applied to data caches when hints about user access patterns are available a priori or when such patterns can be automatically detected. In this paper, however, we focus on sequential access patterns.

3. Methodology and Rationale

3.1 Prefix Caching for Scientific Datasets

With prefix caching, only an initial portion of each dataset is stored in the cache. It allows us to "bootstrap" the data download process with the in-cache prefix. It is a lazy method in the sense that complete retrievals are not initiated until datasets are actually accessed: the cached prefix is served to the client while at the same time the missing suffix is fetched and patched transparently from the original data source. Even then, the retrieved suffix is not permanently stored in the cache.

Prefix caching for scientific data is made feasible due to several facts and trends regarding both data-intensive analysis and file transfer tools. First, as mentioned earlier, datasets are almost always safely archived at an external primary source (tapes, Internet databases, parallel file systems). These relatively stable repositories provide reliable data sources for access-time suffix patching in order to seamlessly patch the suffix of the dataset. Second, scientific datasets, especially raw data archived at shared storage repositories, are usually write-once-read-many. Thus, consistency between the partially cached copy and the primary copy is not a big concern. Third, scientific data analysis often has a sequential access pattern (e.g., in file scans for statistical analysis or mining, time-series visualization processing to generate movies from simulation output, and biological sequence alignments against large sequence databases). Finally, several bulk scientific data movement tools (such as HSI, GridFTP, and LoRS) support the ability to fetch partial files (specified by the starting offset and extent), directly facilitating efficient suffix retrieval.

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 hide the cost of suffix patching, a sufficiently large enough prefix of the dataset should cached. The desired prefix size is determined based on four parameters: the size of the dataset, the in-cache data access rate, the suffix patching rate, and the suffix patching initial startup latency. Consequently, the ideal prefix depends on the internal and external environments. The following model determines the size of the prefix, S_{prefix}, necessary to ensure that the uncached suffix will be fetched in time to deliver the same local access performance as if the entire dataset were cached. It assumes sequential access from the client, which as explained earlier, is often true for scientific data processing.

Suppose the dataset size is S and the in-cache client access rate afforded by the distributed cache is R_{client}. The cost of parallel patching from the external data source is formulated into two parts: the initial size-independent overhead L (which includes the costs of creating the connection, authentication, loading the file from tape systems, etc.), and the size-dependent cost of parallel data transfer at the aggregate rate R_{patch}.

We assume that $R_{patch} < R_{client}$, which means the client fetches data from the local cache faster than directly from the external data source—implying the need for a cache. Given the client access rate, we know how much time we have to patch. The equation below equates client access time and the time of the two components of patching,

$$\frac{S}{R_{client}} = L + \frac{S - S_{prefix}}{R_{patch}}.$$

Solving for the prefix size, we get

$$S_{prefix} = S\left(1 - \frac{R_{patch}}{R_{client}}\right) + LR_{patch}.$$

With this model, the appropriate prefix size can be calculated for each individual dataset, taking into account the external source storing its primary copy. Parameters such as L and R_{patch} for each data source can be stored at the cache manager as a part of the metadata. As the total number of scientific repositories for a dataset is limited, the time and effort required to benchmark and save such parameters should not be significant. In addition, these parameters can be derived from actual dataset imports and suffix patches, enabling the prefix size prediction to adapt to changes in the remote storage systems and in networking hardware/software.

3.2 Collective Download

In this section, we discuss optimizing the parallel prefix patching process to enhance the aggregate patching rate R_{patch}. Our approach is inspired by the collective I/O strategy used in parallel I/O libraries, which aggregate small, non-contiguous I/O requests into large, sequential ones to achieve better file read/write performance. Such an aggregation is necessary since the way data is partitioned in distributed memories often forces processors to issue many disjoint file system requests (e.g., a 2-D array distributed in a BLOCK manner across processors but written in row-major order in a shared file). In parallel I/O libraries, collective I/O is typically done by performing client-side inter-processor communication to exchange data (called *shuffling* or *data reorganization* [30]).

We face a similar problem in suffix patching when the cache stripes data across multiple nodes. For better local sequential access performance, it is preferred that data be striped at a relatively fine granularity (say hundreds of KBs to several MBs). This is so that the I/O and network bandwidth at the participating nodes can be effectively aggregated. On the flip side, for better download performance from external data sources, it is beneficial if each patching node issues a small number of large requests. Note that in the parallel I/O context, "several MBs" is considered a large request size, which is already many times a typical file system's block size. For example, the default aggregated I/O request size is 4MB in the widely used ROMIO implementation of MPI-IO [43, 42]. However, such a request size is unable to exploit today's popular scientific data transfer tools, as we will show in Section 5. This may be attributed to several factors, including the less predictable data transfer bandwidth, the higher per request overhead at certain data sources (such as a tape system), and the fact that protocols/tools such as GridFTP and HSI are designed and tuned for bulk data transfer.

To alleviate the problem, we propose *collective download* that allows the patching node to issue large partial file retrieval requests from the external data sources. Simultaneously, the downloaded data is shuffled locally for a rearranged layout, conforming to the smaller stripe size used in the distributed cache. It is not difficult to pipeline the data shuffling with data download, so the cost of the faster operation is hidden. Further, parallel data download and data shuffling can be interleaved with serving data stripes to the client. Implementation details will be discussed in Section 4.2. Through our experiments, we have found that for large datasets, the retrieval request size should be in the order of hundreds of MBs.

3.3 New Architecture for Desktop Resource Aggregation

Besides its performance advantages, combining prefix caching with collective download enables a new architecture for aggregating unused resources in distributed desktop environments.

Existing resource stealing/aggregation systems have focused on harnessing idle CPU cycles (e.g., Condor [29] and Entropia [14]) or unused disk space (e.g., Farsite [7], FreeLoader [47], and Kosha [12]). With prefix caching, we allow a distributed cache built atop donated disk space to utilize nodes that are willing to share their network bandwidth but not storage space. These nodes will be enrolled as *dedicated patching nodes*, as opposed to storage nodes that contribute persistent storage resources. This helps attract more participants to join the distributed cache, for example, nodes with limited secondary storage space but good network connectivity.

With this architecture, dedicated patching nodes only perform parallel downloads from external data sources, allowing suffix data to be streamed without being stored persistently. Storage nodes, on the other hand, will store stripes of prefix data and their donated space will be managed by a cache replacement policy.

For optimal performance, collective download should be carried out in memory at the patching nodes. As in parallel I/O libraries, data shuffling itself can be conducted incrementally with a relatively small additional buffer space. Therefore, the partial file retrieval request size also indicates the memory space required

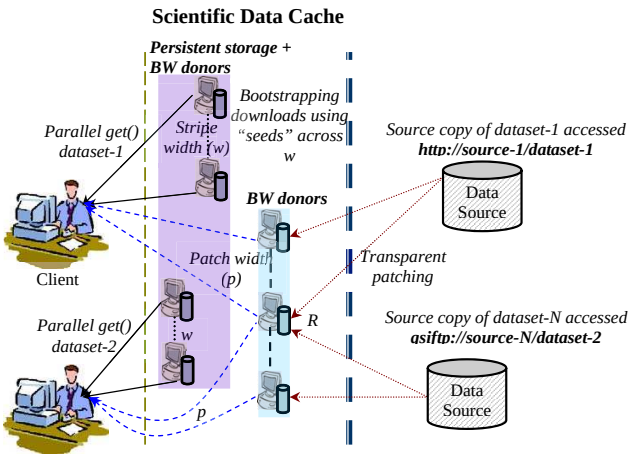

Figure 2. Prefix caching environment with storage nodes and dedicated patching nodes

to buffer the retrieved data. In the resource stealing scenario, this imposes an additional constraint on the file retrieval request size. In a separate study, we have found that when both the native workload on a donated machine and the foreign workload (in this case suffix patching and serving data to the client) are both memory-intensive, the paging caused by memory contention will have a significant impact on both workloads [17]. While it is desirable to maximize I/O request sizes, in our experiments we used 256MB as the largest size to reduce the likelihood of memory contention.

Figure 2 depicts the new resource aggregation architecture enabled by prefix caching. When a partially cached dataset is requested by a client, the storage nodes, which donate both persistent storage resources and transient bandwidths, will serve the cached prefix. Meanwhile the dedicated patching nodes, which donate bandwidths only, will stream the suffix from external data sources.

4. Key Design and Implementation Issues

4.1 Cache Management and Dataset Access

We implemented the prefix caching and collective download approach in the FreeLoader prototype, introduced in Section 2.1. When a partially cached dataset is accessed, the manager passes the chunk distribution map to the client and allows it to start retrieving the cached prefix from the appropriate storage nodes. Meanwhile, p patching nodes are selected, first among those specialized patching nodes, who elect to donate network bandwidth but not disk space. If there are not enough such nodes, the manager will enlist storage nodes too, preferably those who are not serving data at the moment. Since data is striped in a round-robin manner, the manager can decide the chunk distribution map for the suffix and communicate it to the client. The suffix retrieved by the patching nodes will be discarded after the client finishes the access.

When a dataset is initially imported into the cache, we choose between two strategies depending on the cache space availability. When there is enough empty cache space to accommodate the new dataset, the dataset is cached in its entirety. When cache eviction is needed, however, only a prefix of the new dataset will be striped to the storage nodes. The size of this prefix is calculated using the model given in Section 3.1. This way, a small working set will automatically result in the caching of entire datasets.

Our cache eviction is based on LRU, but extended for prefix caching. More specifically, each dataset is evicted from its tail. If there are datasets whose cached part exceeds the prefix size necessary (again according to the prefix size prediction model), we select victims using LRU and evict chunks from each dataset to keep only the desired prefix size. When no such datasets exist and still more chunks are needed, we evict dataset (the cached prefix) victims, one after another, again by LRU. Note that even a dataset whose cached prefix is shorter than the desired level, when accessed again, can be served with our patching scheme, though performance penalty may be observed by the client.

4.2 Collective Download and Patching Implementation

We have implemented collective download using a combination of large downloads and concurrent shuffling of the downloaded data. As described in Section 3, the large downloads are performed to increase the downloading rate from external data repositories. The resultant data is required to be shuffled locally—among the p patching nodes—according to local striping policies to optimize client accesses to the cache. For instance, FreeLoader uses a default 1MB chunk as the basic unit of striping, whereas remote downloads perform best at a granularity of hundreds of megabytes.

At the beginning of each collective download operation, a *session* is established between the patching nodes and the external data source. Many remote sources assume downloads occur in an interactive session that includes authentication, such as GridFTP servers using UberFTP client [5] and HPSS using HSI [18]. In our implementation, the patching nodes use Expect [28], a tool specifically geared towards automating interactive applications, to establish and manage these interactive sessions. We instrumented the FreeLoader patching framework with Expect so that authentications and subsequent partial retrieval requests to a remote source can be performed over a single stateful session. This implementation mitigates the effects of authentication and connection establishment by amortizing these large, one-time costs over multiple partial file retrieval requests.

During shuffling, each patching node sets up TCP connections to their peers involved in the patching process, to send and receive chunks from the downloaded data. With a patching width of p, approximately $(p-1)/p$ of a patching node's downloaded data has to be sent to its peers. Therefore, for a reasonable patching width (one that at least matches the cache's stripe width to ensure good client access throughput), a substantial amount of data is shuffled. Thus, it is important to optimize the performance of shuffling and to efficiently overlap it with the download. In our implementation, we use separate threads to perform the download and shuffling (which is not the case in popular collective I/O libraries due to MPI thread-safety concerns). As downloaded data arrive, nodes immediately start re-distributing chunks to other patching nodes. They will concurrently serve chunks to the client if requested by the latter. The shuffling process is accomplished in memory to expedite data redistribution and to motivate bandwidth-only contributions from donors.

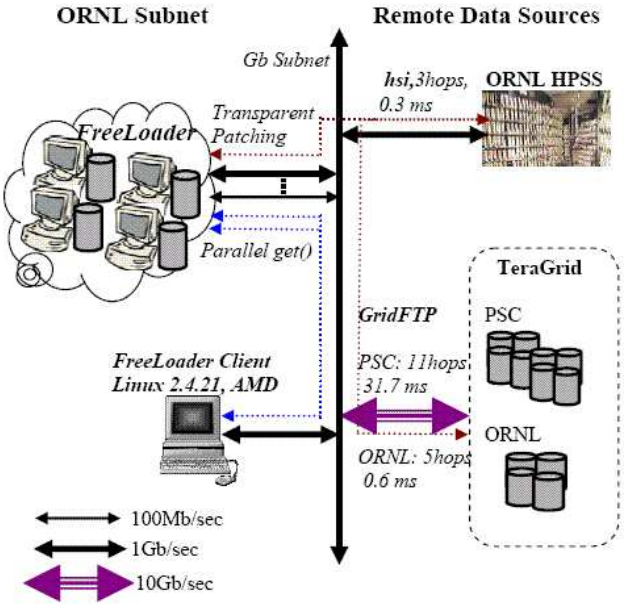

Figure 3. Testbed setup

5. Performance Results

This section presents performance results measured from our FreeLoader testbed using multiple external data sources, as well as from a trace-driven simulation.

5.1 Testbed Configuration

Our testbed (Figure 3) depicts a scientist's HPC research environment with a powerful, well-connected local client machine, with access to external data sources such as parallel file systems and mass storage systems.

We installed the FreeLoader storage cache in this setting to study its ability to patch datasets from external data sources in a transparent manner. Our testbed spreads across Oak Ridge National Laboratory (ORNL) and the TeraGrid (a Nationally deployed Grid infrastructure). It consists of the following components.

1. Remote data sources where scientists store and/or share primary copies of their datasets (identified by the protocol name and the location of the source). The HPSS [15] archival storage system at ORNL (HPSS-ORNL), with 365PB of tape storage accessed through HSI. GridFTP servers, enabling access to parallel file systems on the TeraGrid sites, ORNL (TeraGrid-ORNL) and Pittsburgh Supercomputing Center (TeraGrid-PSC). We used the UberFTP client interface to access the GridFTP servers. Figure 3 also shows the connectivity of these remote data sources to the ORNL subnet, which results in varied patch bandwidth.

2. The FreeLoader cloud at ORNL contains aggregate storage of 0.5TB on 13 storage nodes (donating 7-60GBs each) and one manager. These donated machines have dual Pentium III, Linux 2.4.20-8 kernel, and mostly 100Mb/sec Ethernet connection. The storage nodes are equipped to patch from the remote sources using appropriate protocols as well.

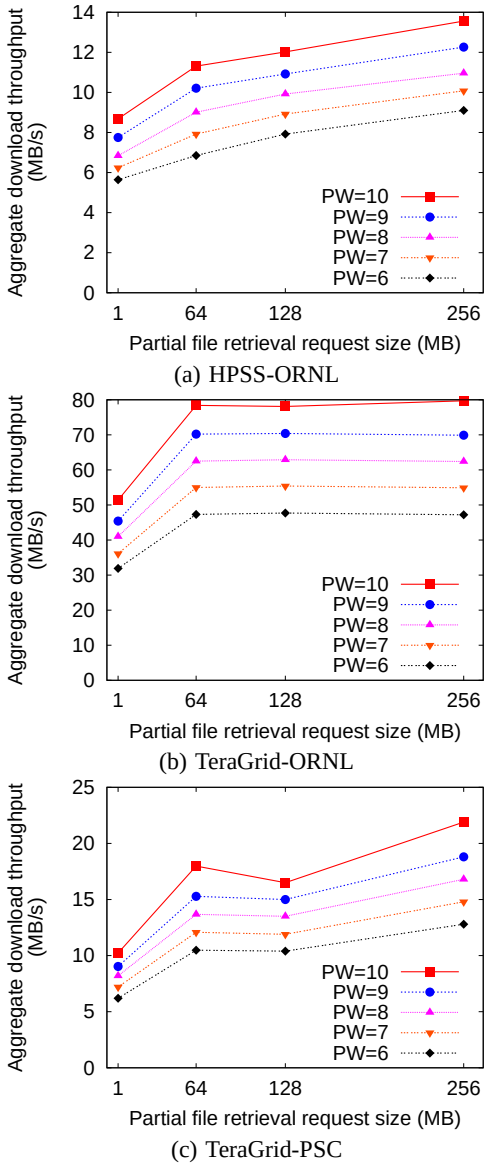

Figure 4. Parallel download rates

3. A client machine at ORNL with Dual AMD Opteron, Linux 2.4.21 and GigE, running the FreeLoader client component. It is at most five hops away from any of the storage nodes in the FreeLoader cloud.

In this paper, all the experiments performed with the above testbed use a dataset size of 2.5GB. The results reported are the averages of 3 or more tests (with no significant variance observed). Further, for HPSS, we consider "cold" accesses (i.e., retrieving a dataset from the tape). Even though HPSS has high-speed disk caches atop tapes, such caches can only accomodate a small portion of the tape capacity and are typically shared by a large group of users.

5.2 Collective Download Performance

Figure 4 shows the parallel download rate from each of the

	Download	Download & shuffle		Client access	
HPSS	13.6	12.3	-9.6%	11.7	-4.9%
Tera-ORNL	79.7	75.1	-5.8%	74.7	-1.3%
Tera-PSC	21.9	20.2	-7.8%	20.0	-1.0%

Table 1. Collective download performance, MB/s (patching width = 10, partial file request size = 256MB)

external data sources. The results are collected using a chunk size of 1MB (which is also the stripe size) when storing the dataset in the FreeLoader space.

For each data source, we report the impact of varying two parameters: the remote I/O request size and the patching width (PW, the number of patching nodes performing parallel patching). Here the remote I/O request size is the size of each contiguous partial file retrieval request made by a patching node. For parallel downloading, the file is partitioned among the patching nodes in a round-robin way, at the unit of the remote I/O request size. The patching nodes will perform data shuffling to re-stripe the downloaded data among themselves to have a stripe size of 1MB. Therefore, with a remote I/O request size of 1MB, no data shuffling is necessary.

Figure 4 clearly indicates the need for collective download. For all three data sources, significant improvement in the aggregate download performance can be achieved by using a larger request size. For example, at a patching width of 10, the throughput increase ranges from 55% (TeraGrid-ORNL) to 114% (TeraGrid-PSC) when using a request size of 256MB instead of 1MB. This means collecting the target file requests from patching nodes into fewer larger requests delivers much higher overall download performance. The desired request size, however, varies between different data sources. For TeraGrid-ORNL, it seems a request size of 64MB suffices. For the other two data sources, a larger request size is desirable, although in our implementation we keep the request size below 256MB for memory space concerns, as explained in Section 3.3. This result suggests that using a larger request size might be beneficial for a high-latency protocol/storage system. Benchmarking results shown in Figure 4 can be also saved as part of the management metadata to select a sufficiently large partial file request size for an individual data source, while minimizing the memory usage at patching nodes.

For all the data sources, parallel download appears to scale with patch width. The aggregate throughput steadily increases as more nodes are involved in the download. Limited by our testbed resources, we stopped at a width of 10, to match the best FreeLoader stripe width observed from our experiments (Figure 1). With a larger FreeLoader deployment and more patching nodes, it may be helpful, especially for the slower data sources, to have a patching width that is a multiple of the desired FreeLoader stripe width, to work in groups on disjoint segments of the suffix.

Table 1 shows the performance of collective download, with a patching width of 10. For each data source, we plot the aggregate throughput for three types of operations: (1) "download," where each patching node simply download its assigned portions of the dataset, (2) "download and shuffle," where the patching nodes also exchange their downloaded data, and (3) "client access," where the patching nodes serve the downloaded and shuffled data to the client, with a stripe size of 1MB.

Data Source	HPSS-ORNL	Tera-ORNL	Tera-PSC
R_{client} **(MB/s)**	52.2	52.2	52.2
R_{patch} **(MB/s)**	7.6	42.0	10.8
L **(s)**	31.4	3.0	3.9
Predicted ratio	95.0%	24.6%	81.0%

Table 2. Prefix prediction parameters and results

The results demonstrate that collective download can be implemented efficiently. When data shuffling is added, it overlaps well with the parallel download and the maximum decrease in the aggregate throughput is 9.6% (HPSS-ORNL). Further, when the patching nodes have to serve data to the client, the performance reduction from the previous step, "download and shuffle", is even smaller. Note that the client access rate here indicates the user-perceived cost of retrieving the entire dataset from the external source (with a 0% prefix cached).

5.3 Prefix Size Prediction Model Verification

In this section, we verify the prefix size prediction model given in Section 3.1. Also, in this group of experiments, we test the new architecture described in Section 3.3, where two groups of donated machines assume the roles of storage nodes and patching nodes respectively. Limited by our testbed, we assigned 6 nodes to be storage nodes, who contribute persistent storage resources and store the cached prefix of datasets. Another 6 nodes are assigned to be patching nodes, who contribute bandwidths.

First, we collect the required parameters from the above test configuration. Table 2 summarizes the parameter values and the predicted prefix ratios. The client access rate R_{client} measures the client's throughput of accessing an in-cache dataset (striped at 6 storage nodes) and is therefore independent of the data sources. R_{patch} is measured using 6 patching nodes, who perform downloading plus shuffling. L is measured by timing the overhead of retrieving a 0-byte file from the data sources. With these parameters, we calculate the predicted prefix ratio (percentage of the dataset size) for each data source.

Figure 5 verifies the above predicted prefix ratios against the measured client access rate at a series of prefix ratios. At each given prefix ratio, the prefix is stored at the 6 storage nodes, and the suffix is patched by the 6 patching nodes. The three curves illustrate the impact of the increasing prefix ratio on the client access rate. As expected, the rate gradually approaches the maximum rate (the rate measured with a prefix ratio of 100%, i.e., with the entire dataset cached). Ideally, the predicted prefix ratio (indicated by the vertical lines) should be at the saturation point of each curve.

From Figure 5, we can see that our prediction model works fairly well. For TeraGrid-ORNL, the model predicts a prefix ratio of about 25%, while the actual saturation point is located between 30% and 35%. For TeraGrid-PSC, the model predicts 81%, while the actual saturation point is between 80% and 85%. For HPSS-ORNL, the model predicts 94% while the actual saturation point is between 95% and 100% (which indicates that for slow data sources with a high predicted prefix ratio, caching the entire dataset is a good choice). Even with the largest error (TeraGrid-ORNL), the client access rate achieved is 90% of the saturated performance.

The result does show that our prediction model appears to be

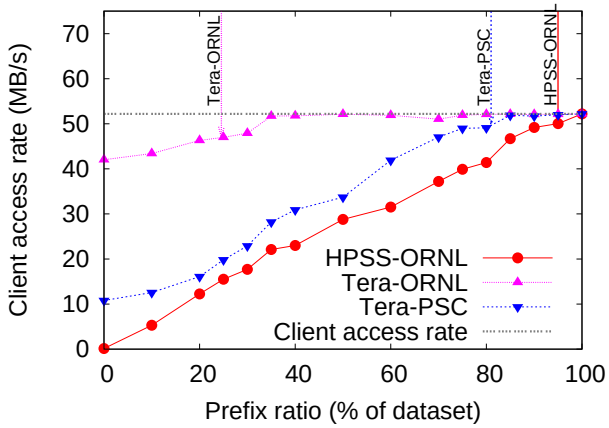

Figure 5. Client access rates at varied prefix ratios. The vertical lines mark the model predicted prefix ratio for the data sources.

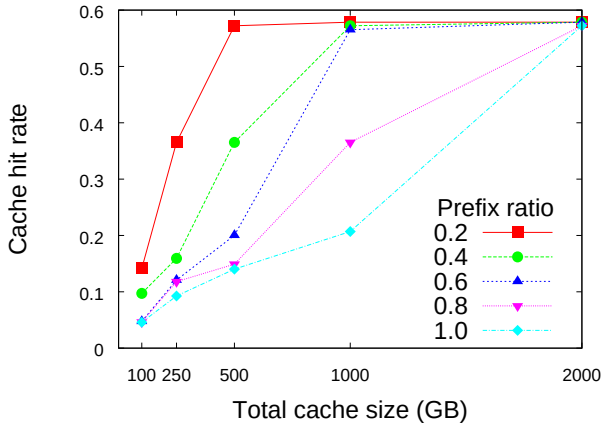

Figure 6. Cache hit rates at varied prefix ratios using the Jasmine trace

consistently optimistic. Our speculation is that the benchmarking of R_{patch} may have obtained higher numbers since the tests were conducted on the patching nodes without having the storage nodes perform data serving simultaneously. This may result in a lower local network traffic than in the model verification test. However, the assignment of patching nodes is dynamic anyway, hence determining a perfect prefix size according to the accurate patching rate does not seem practical.

The prefix size is large for high-latency data sources and small for low-latency sources. However, if a set of clients can be satisfied with 80% of the full bandwidth or cannot sustain maximum possible bandwidth, we can cache 75% of the dataset (say, for HPSS-ORNL) and still not impact client perceived throughput.

5.4 Simulation Results: The Impact of Prefix Caching on Cache Hit Rate

We perform a trace-driven simulation to evaluate the impact of prefix caching on the cache space utilization. Our simulator takes as input a dataset access trace, and manages the cache using the caching strategies described in Section 4.1. The access trace we used is a real-world trace containing the Jefferson Laboratory researchers' access log to the high-energy physics data hosted at the Jefferson Lab Asynchronous Storage Manager (JASMine). Due to its large size, we filtered the trace to include datasets with a size of at least 2GB, considering that users will be more interested in storing larger datasets in a distributed cache. The filtered trace spans 19.1 days and has 4000 entries accessing 1686 unique datasets.

Figure 6 plots the cache hit rates at different prefix ratio levels. To illustrate the impact of the prefix ratio, the same prefix ratio is applied to every dataset (a cache storing datasets from different data sources will apply different prefix ratios to them). The "1.0" curve corresponds to the original caching strategy storing entire datasets. The results reveal that with scientists' real access patterns, prefix caching may result in a dramatic improvement in cache hit rate, and allows a cache to reach the maximum performance allowed by the access pattern with a much smaller total cache space. Considering the data sources used in our testbed, faster sources (such as TeraGrid-ORNL) that allow a prefix ratio of around 30% will observe an improvement between the "0.2" curve and the "0.4" curve in cache hit rate (up to 308% and 176% with a prefix ratio of 20% and 40% respectively). Even a significantly slower data source, TeraGrid-PSC, will receive an up to 76% improvement in hit rate with a prefix ratio of 80%. Finally, the expected cache hit rate improvement will grow as the high-end computing network infrastructure and file transfer tools are enhanced.

6. Conclusions

This paper demonstrates the design and implementation of prefix caching plus collective download for caching large scientific datasets. We have shown that prefix access can be effectively overlapped with suffix patching from external data sources. Meanwhile, by merging small remote file retrieval requests into larger ones and performing local data shuffling, we can achieve high performance in both dataset downloading performance and local cache accesses. The combination of the above techniques obviates the need for always caching large datasets in their entirety, and improves the overall cache space utilization. We also demonstrate that the desired prefix size for individual data sources can be predicted fairly accurately using an analytical model. Further, we argue that our proposed solution enables new resource sharing models. We summarize our contributions in this paper as follows.

- We proposed novel novel techniques to overlap remote I/O with cache I/O and demonstrated the usefulness of collective I/O in bulk data transfer from high-end scientific data repositories.
- We designed and built the prototype prefix caching architecture in the context of our FreeLoader collaborative desktop cache environment.
- We performed both experimental as well as simulation studies, using multiple scientific data repositories and a real-world scientific data access trace, to evaluate our proposed

approaches and model.

7. Acknowledgment

This work is supported in part by a DOE ECPI Award (DE-FG02-05ER25685), an NSF CAREER Award (CNS-0546301), an IBM UPP award, a DOE contract with UT-Battelle, LLC (DE-AC05-00OR2275), and Xiaosong Ma's joint appointment between NCSU and ORNL. The authors thank John Cobb and Greg Pike for access to TeraGrid resources.

8. References

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

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

[3] Ccsm-community climate system model. http://www.ccsm.ucar.edu, 2006.

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

[5] Ncsa gridftp client. http://dims.ncsa.uiuc.edu/set/uberftp/index.html, 2006.

[6] S. Acharya and B. Smith. Middleman:a video caching proxy server. In *Proceedings of 10th international workshop on network and operating system support for digital audio and video (NOSSDAV)*, 2000.

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

[8] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link. The Globus Striped GridFTP framework and server. In *Proceedings of Supercomputing '05*, 2001.

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

[10] A. Bharambe, C. Herley, and V. Padmanabhan. Analyzing and Improving a BitTorrent Network's Performance Mechanisms. In *Proceedings of INFOCOM 2006*, 2006.

[11] R. Bordawekar, J. Rosario, and A. Choudhary. Design and evaluation of primitives for parallel I/O. In *Proceedings of Supercomputing '93*, 1993.

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

[13] J. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of the ACM SIGCOMM Conference*, 1998.

[14] B. Calder, A. Chien, J. Wang, and D. Yang. The Entropia virtual machine for desktop grids. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, 2005.

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

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

[17] Vincent W. Freeh, Xiaosong Ma, Sudharshan S. Vazhkudai, and Jonathan W. Strickland. Controlling impact while aggressively scavenging idle resources. Technical Report TR-2006-7, North Carolina State University, Raleigh, NC, February 2006. In submission.

[18] M. Gleicher. HSI: Hierarchical storage interface for HPSS. http://www.hpss-collaboration.org/hpss/HSI/.

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

[20] 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.

[21] S. Gruber, J. Rexford, and A. Basso. Protocol considerations for a prefix-caching proxy for multimedia streams. http://www9.org/w9cdrom/349/349.html.

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

[23] S. Jin, A. Bestavros, and A. Iyengar. Network-aware partial caching for internet streaming media delivery. *ACM/Springer Multimedia Systems*, 9(4), 2003.

[24] M. Kallahalla and P. J. Varman. PC-OPT: Optimal offline prefetching and caching for parallel I/O systems. *IEEE Transactions on Computers*, 51(11), 2002.

[25] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, November 1994.

[26] J. Lee, X. Ma, R. Ross, R. Thakur, and M. Winslett. RFS: Efficient and flexible remote file access for MPI-IO. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2004.

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

[28] D. Libes. The expect home page. http://expect.nist.gov/, 2006.

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

[30] J. May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann Publishers, 2001.

[31] N. Nieuwejaar and D. Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, 1997.

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

[33] V. Padmanabhan. Using Predictive Prefetching to Improve World Wide Web Latency . In *Proceedings of ACM SIGCOMM*, 1996.

[34] J. Plank, S. Atchley, Y. Ding, and M. Beck. Algorithms for high performance, wide-area distributed file downloads. *Parallel Processing Letters*, 13(2), 2003.

[35] 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.

[36] P. Rodriguez, A. Kirpal, and W.E. Biersack. Parallel-access for Mirror Sites in the Internet. In *Proceedings of IEEE INFOCOM*, 2000.

[37] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (rtsp). http://www.ietf.org/rfc/rfc2326.txt, 1998.

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

[39] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, 1995.

[40] S. Sen, J. Rexford, and D. Towsley. Proxy prefix caching for multimedia streams. In *Proceedings of the IEEE INFOCOM Conference*, 1999.

[41] 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.

[42] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999.

[43] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[44] B. Tierney, D. Gunter, J. Lee, and M. Stoufer. Enabling network-aware applications. In *Proceedings of the IEEE High Performance Distributed Computing conference*, 2001.

[45] B. Tierney, J. Lee, M. Holding, J. Hylton, and F. Drake. A network-aware distributed storage cache for data intensive environments. In *Proceedings of the IEEE High Performance Distributed Computing conference (HPDC-8)*, 1999.

[46] S. Vazhkudai. Distributed downloads of bulk, replicated grid data. *International Journal of Grid Computing*, (2), 2004.

[47] 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.

[48] E. Weigle and A. Chien. The composite endpoint protocol (CEP): Scalable endpoints for terabit flows. In *Proceedings of the IEEE Conference on Cluster Computing and the Grid*, 2005.