J Grid Computing
DOI 10.1007/s10723-009-9122-7

Improving Data Availability for Better Access Performance: A Study on Caching Scientific Data on Distributed Desktop Workstations

Xiaosong Ma · Sudharshan S. Vazhkudai · Zhe Zhang

Received: 25 February 2009 / Accepted: 8 July 2009
© Springer Science + Business Media B.V. 2009

Abstract Client-side data caching serves as an excellent mechanism to store and analyze the rapidly growing scientific data, motivating distributed, client-side caches built from unreliable desktop storage contributions to store and access large scientific data. They offer several desirable properties, such as performance impedance matching, improved space utilization, and high parallel I/O bandwidth. In this context, we are faced with two key challenges: (1) the finite amount of contributed cache space is stretched by the ever increasing scientific dataset sizes and (2) the transient nature of volunteered storage nodes impacts data availability. In this article, we address these challenges by exploiting the existence of external, primary copies of datasets. We propose a novel combination of *prefix caching*, *collective download*, and *remote partial data recovery (RPDR)*, to deal with optimal cache space consumption and storage node volatility. Our evaluation, performed on our FreeLoader prototype, indicates that prefix caching can significantly improve the cache hit rate and partial data recovery is better than (or comparable to) many persistent-data availability techniques.

Keywords Desktop grids · Storage scavenging · Scientific data

This work builds on several of our previously published studies. While we included extended introduction of the FreeLoader framework [57, 58] as well as our combination of prefix caching and collective download techniques [33], this paper proposes the novel RDPR technique as well as a new striping-enabled cache management algorithm. We also discussed methods to combine RDPR with prefix caching and collective download techniques. Overall, more than 60% of the manuscript's content has not been previously published.

X. Ma · Z. Zhang (✉)
North Carolina State University,
Raleigh, NC, USA
e-mail: zzhang3@ncsu.edu

X. Ma · S. S. Vazhkudai
Oak Ridge National Laboratory,
Oak Ridge, TN, USA

1 Introduction

The ever increasing disk capacity combined with decreasing cost per gigabyte has created larger amounts of unused disk space on desktop computers. Studies have shown that, on average, half of the disk space on desktop workstations is free and that this fraction is only increasing as disks become larger [2, 15]. As a result, numerous distributed file systems exist that aggregate desktop storage, both in the local [2, 9] and the wide-area

Published online: 16 July 2009

[16, 17] environments. These distributed storage systems aim at delivering reliable and persistent storage with data availability levels commensurate to centralized file servers.

Scientists are faced with rapidly increasing amounts of data collected from physical experiments, observatory instruments, and computer simulations [23, 53]. Scientists prefer to archive their datasets at shared repositories, such as the mass storage systems (e.g., HPSS [13]) and data centers (e.g., SDSS [47]). 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 and fault tolerance. However, scientific computing workflows are often distributed, requiring further analysis and visualization on scientists' local workstations. For instance, the resulting data may have to be visualized to glean insights or to add tweaks to the processing using a feedback loop. Consequently, scientists' personal workstations are an integral part of their job workflow. Personal workstations possess indispensable display and interactive processing capabilities, but often lack sufficient storage capacity or bandwidth. Despite the growing PC storage space, scientific dataset sizes can easily overwhelm a single workstation. The upshot is that end-users are not able to store all their data locally and often resort to expensive, repeated wide-area data movement.

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 [55]. Aggregated storage within the scientist's LAN provides a low-cost alternative to buying and maintaining dedicated storage servers or clusters. In addition to space aggregation, striping data on multiple nodes, each of which contributing a fraction of its unused storage, also enables bandwidth aggregation. This results in better overall access throughput. These techniques have been shown to be viable in our previous work on FreeLoader [57, 58], an aggregate storage infrastructure for large scientific datasets.

Scientists normally perform a variety of processing tasks on groups of datasets (*e.g.*, time-series results generated from a recent parallel simulation) for a limited period of time, typically days or weeks. Meanwhile, colleagues working in the same research area tend to have shared interests on common data [25]. Therefore, the collective free space is suitable to be positioned as a *cache* for remote data. This reduces repeated wide-area data migration costs for scientists reusing hot datasets and helps achieve better access throughput than retrieving data from one's local hard disk. Compared to quota-based resource allocation, common in general-purpose file systems, a cache replacement policy allows for better capacity management. It accommodates large datasets and achieves better overall space utilization.

However, existing systems cache scientific datasets in their entirety, while scavenged storage systems have to be particularly careful about space and bandwidth usage. We show that storing large datasets in their entirety 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 [33]. 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.

Another challenge posed by caching data on distributed, unreliable workstations is data availability. The storage nodes in our target environment are ordinary workstations that are subject to shutdowns, reboots, and failures. Striped data placement, combined with the sequential data access pattern, complicates caching when one or more storage nodes fail and recover dynamically. In other words, "partial cache miss" has to be dealt with.

Existing work on storage systems has addressed the data availability in distributed and unreliable

environments by adding redundancy. However, data redundancy reduces the effective cache size, which *decreases* the overall data availability. Scientific data caches built atop distributed workstations, on the other hand, can take advantage of the existing data redundancy external to the local network environment. Scientific datasets usually have primary, immutable copies archived at external data repositories. These include mass storage systems, shared file systems at supercomputers, data centers (such as SDSS [47]), and websites (such as the NCBI Genbank [34]). Unlike personal files stored on the general-purpose file systems, data availability guarantee for the local copies of scientific datasets is not a crucial requirement. Data availability, in such cases, translates into a *performance* issue: in response to a data miss—caused by new accesses, cache eviction operations, or donor node failures—remote accesses are conducted to fetch the missing data portions.

In this paper, we investigate the performance impact of data availability in distributed caches built on unreliable workstations. More specifically, we consider our major contributions as below.

- We have designed and implemented cache management strategies that handle striped datasets and the failure/recovery of individual storage nodes.
- We proposed *RPDR (remote partial data recovery from external repositories)*, an approach that reduces the partial data miss penalty in a distributed, dynamic, and unreliable cache. We further augmented RPDR with two techniques: prefix caching and collective downloads. We also investigated two redundancy-based approaches (replication and erasure coding) in the data striping, distributed caching context.
- We performed trace-driven simulation to evaluate the effectiveness of the above three data recovery approaches. We co-played real-world scientific data access traces with real-world node availability traces, and computed the data access cost using partial data recovery performance parameters measured from an aggregate storage system implementation. The results revealed that our proposed RPDR approach and a combination of RPDR and erasure coding perform the best in most of the test cases, with a performance advantage of up to 38% and 54% over the redundancy-based approaches for the two data access traces respectively.

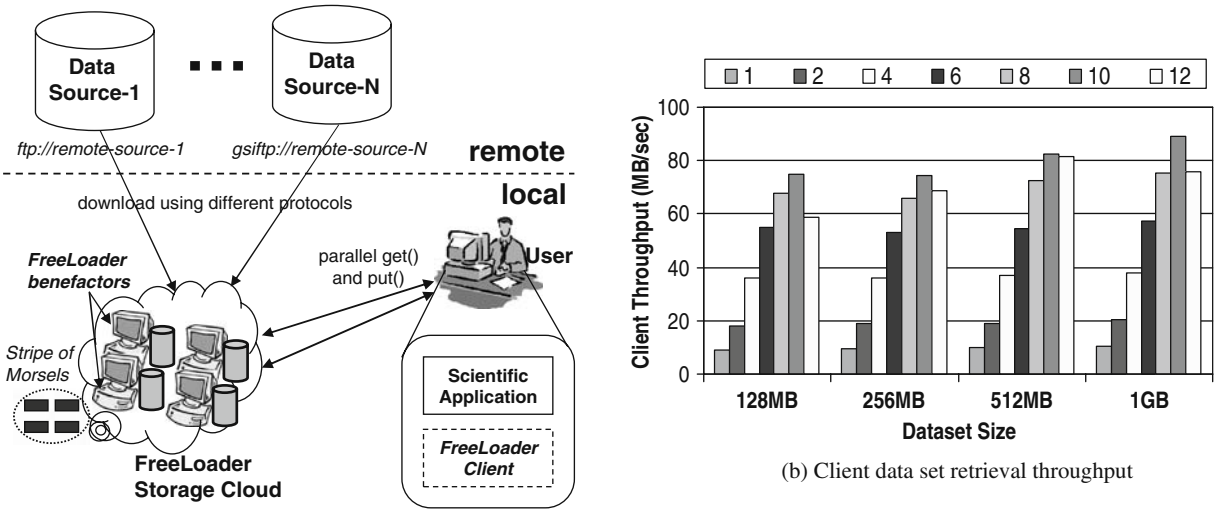
The individual elements of this study, such as partial data caching, striping, fault-tolerance in distributed storage, and patching data from remote sites, are themselves not new. However, the key novelty of our work lies in the *combination* of these elements, *i.e.*, the examination of recovery strategies for striped data cached on unreliable nodes. This combination of the above techniques is motivated by real needs of scientific data processing, and evaluated with scientific workloads.

2 Background

2.1 FreeLoader: An Aggregate Storage System Prototype

Our simulation-based study is motivated by, and parameterized with the FreeLoader storage aggregation prototype [57], built atop scavenged storage resources. It aggregates unused disk space in a LAN setting into a unified cache and scratch space for storing scientific datasets. In our prototype, participating workstations donate idle disk space and data is striped on multiple such storage nodes in fixed-size chunks.

A dedicated manager node maintains metadata such as node status, chunk distribution, and dataset attributes including the primary copy location (URI, protocol/tool used to retrieve/import the dataset, and authentication related metadata). Since this system is positioned to cache popular scientific datasets with primary copies at external data repositories, aforementioned metadata concerning dataset import enables transparent recovery of datasets from their primary copies. Further, this consumes no additional metadata storage than what is required to store a dataset. Figure 1a illustrates the system design.



(a) Free Loader Environment

(b) Client data set retrieval throughput

Fig. 1 The FreeLoader environment overview and sample performance (**a**, **b**)

Figure 1b demonstrates FreeLoader's data access performance by showing the client-side dataset retrieval rates 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 100 Mbit/s). The client's Gb/s network configuration is meant to capture scientists' high-end desktop workstations with good connectivity for scientific data processing. The storage nodes' configuration (100 Mb/s connectivity) is meant to show how FreeLoader can reap significant throughput even in the face of modest connectivity. Figure 1b shows that in such an environment, the benefits due to storage aggregation through striping are manifold. First, it provides larger capacity by utilizing other workstations' unused disk space. Second, it also provides a performance benefit by aggregating I/O and network bandwidth available at the distributed machines. Third, even when a user can accommodate her datasets at her own workstation, storing such data in an aggregated cache (such as the one prototyped) will deliver an I/O throughput (as high as 88 MB/s with a stripe width of 10) significantly better than local disk I/O rate (typically 30–50 MB/s). Finally, our prior results revealed that striping is also an effective way to reduce the impact on space donors' native workloads.

2.2 Related Work

In this section, we will survey related work with reference to prefix caching, collective download and data availability.

Prefix caching [24, 49] techniques have been proposed and implemented for multimedia streaming protocols such as the IETF Real-Time Streaming Protocol [46] that is built on HTTP. Our work applies prefix caching to scientific data storage/access, and leverages the parallel transfer capability offered by tools such as GridFTP [3, 4], HSI [22], and LoRS [39].

Middleman cache for video files on the Web is similar to our effort [1]. 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 [26]. We use similar information to determine the ideal prefix size.

Scientific data caches like IBP [40] and DPSS [56] 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. 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.

Our collective download approach resembles the collective I/O technique extensively studied in the parallel I/O field and widely used in parallel simulations [8, 28, 35, 48, 54]. 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 [29, 30], 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.

Predictive prefetching is a widely researched topic to improve WWW as well as cache performance [27, 37]. 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.

Data availability and fault tolerance in storage and file systems have been studied extensively. Creating hardware/data redundancy has been a traditional approach to address these problems. Common hardware solutions include, among others, many versions of RAID systems [38] and node failover strategies [11, 45]. Several software approaches to data availability have also been deployed, especially in distributed and peer-to-peer environments. In particular, replication is widely used and investigated [2, 6, 7, 9, 12, 21, 51]. Similarly, there are many existing studies on erasure coding, and an increasing interest in deploying it in storage and peer-to-peer systems [10, 41–43]. This paper studies these existing mechanisms in a distributed cache with data-striping, and compares them with a new data recovery method that does not rely on local data redundancy. Our trace-driven evaluation, based on workstation availability data in a LAN setting, complements a previous study performing theoretical comparison [59].

Our work is also related to several areas of research on caching. Our proposed scientific data caching can be viewed as extended cooperative caching [14, 18, 20, 44]: it pools secondary storage in a LAN environment to reduce access misses that require wide-area data transfers from external data sources. In addition, datasets in the cache are likely to be from different external data repositories. Therefore, they are likely to have different downloading costs and may benefit from cost-aware caching [19, 60]. Again, our effort complements the above work and recent research in scientific data caching [36] by investigating the combination of striping and caching on unreliable distributed workstations.

This paper targets distributed caches built by amassing donated disk space, and is closely related to storage aggregation systems [2, 9] and more generally, resource scavenging systems [32, 50]. However, existing storage aggregation systems are typically designed to act as general-purpose file systems and require high data availability. For instance, both FARSITE [2] and Kosha [9] replicate each dataset at least three times. In contrast, our system is positioned as a best-effort cache for better data access performance. In our target scenario, "six nines" of data availability is not required. We treat both events—losing data to cache eviction or to node failure—as an acceptable norm that our system will respond to.

Downloading distributed chunks of data is a common practice to expedite data accesses. This has been combined with erasure coding to be more fault tolerant (e.g., OceanStore [17], Digital Fountain [10]). As mentioned earlier, prefix caching of Internet media files has been widely deployed and used. Striping or performing collective I/O to improve performance is a well accepted norm in the parallel I/O community. However, the amalgamation of several aforementioned techniques—into a novel, partial data

recovery scheme—is unique in this paper. Further, this is applied to the emerging problem of caching bulk scientific data.

3 Striping-Enabled Cache Management

To store datasets in a striped fashion on the aggregate storage, we manage the entire space as a single cache. This section presents our striping algorithm.

Striping is dependent on the space contributions on the individual storage nodes. An appropriate *stripe width* (*i.e.*, the number of storage nodes a client communicates with concurrently to retrieve the dataset) allows the client to fully utilize its network bandwidth, and allows the aggregated storage system to reduce metadata management, distribution costs, and impact on donor nodes. Successful striping of incoming datasets implies the ability to distribute data across a desired stripe width as uniformly as possible. This ensures good retrieval rates as well as a fair usage of space across all donors. Therefore, we adopt a greedy algorithm that sorts the current available space on storage nodes in descending order, and performs striping on w storage nodes, where w is the target stripe width. The sorting-striping process is repeated if one of the storage nodes (the one with the smallest available space) runs out of space before the dataset is completely striped. Note that when there are fewer than w storage nodes with space available, a smaller stripe width has to be adopted.

Below, we sketch the $GreedyStriping(D, w, N)$ algorithm, where D is the dataset to be striped, w is the target stripe width, and N is the set of storage nodes participating in the aggregated storage system. D has $length(D)$ chunks. Each storage node $N[i]$ $(1 \leq i \leq |N|)$ has $avail(N[i])$ chunks of currently available space. When *GreedyStriping* is performed, the sum of $avail(N[i])$ has to be greater than or equal to $length(D)$. Otherwise, cache eviction will be carried out first.

We perform cache eviction at the granularity of datasets and within datasets, at the granularity of chunks. When cache replacement is needed, we calculate how many chunks must be reclaimed to store the incoming dataset. We then select victim dataset(s) from those cached, using a cache management strategy such as LRU. Within each victim dataset, the eviction of chunks starts from the tail end since most large scientific data accesses are sequential. Therefore, if only part of the last dataset needs to be evicted, a prefix of that dataset will still remain in the cache. For each data chunk evicted, we increase the number of available chunks for the corresponding storage node by one. Note that the status of the concerned storage nodes are checked during the replacement process: if a chunk—marked for eviction—resides on a storage node that is currently unavailable, the chunk will be marked as "evicted" in the manager metadata. The storage node needs to reconcile (synchronize metadata) with the manager once it comes back to service. The space occupied by these evicted chunks on a returning node will be considered as "available". In this manner, it is ensured that at the end of eviction, the number of *available* chunks on a subset of storage nodes is commensurate to the space required by the new dataset. Finally, the aforementioned *GreedyStriping* algorithm will be applied to stripe the new dataset on to these storage nodes.

Algorithm 1 $GreedyStriping(D, w, N)$

```
chunk = 0
while chunk < length(D) do
  sort N by value of avail in descending order
  nodes = number of storage nodes with avail > 0
  width = min(nodes, w, length(D) − chunk)
  rounds = min(⌊(length(D)−chunk)/width⌋,
                avail(N[width]))
  for i = 1 to rounds do
    for j = 1 to width do
      chunk=chunk + 1
      assign chunk D[chunk] to storage node N[j]
    end for
  end for
end while
```

With our striping-enabled cache management strategy, there are three types of cache misses. (1) *Total miss*: This occurs when the dataset in question has never been imported, has been evicted in

its entirety, or all the storage nodes hosting it are currently unavailable (which is not very likely with a reasonable stripe width). (2) *Partial miss*: This happens when the dataset is cached, but some of the storage nodes hosting it are unavailable. (3) *Tail miss*: This occurs when the dataset is cached, all the storage nodes hosting it are available, but a suffix of it is missing due to cache eviction. In a subsequent section, we will discuss the cost of these misses in conjunction with several data recovery methods studied in this paper.

4 Recovery of Distributed, Striped Data

Striping data over nodes with transient availability often causes a cache hit to become a partial cache miss, when one or more storage nodes, hosting segments of a cached dataset, fail to serve requested data. In the storage aggregation context, nodes are individually-owned PCs donating a portion of their local disk space. Hence collective node availability cannot be scheduled or planned. As a result, sudden benefactor shutdown/crash/withdrawal might render the distributed storage system with little to no time for corrective measures such as data relocation.

In this paper, we examine three data recovery strategies. First, we explore direct partial data recovery from external data sources. We then consider two existing alternatives, namely replication and erasure coding, and discuss their functioning in a distributed, unreliable cache. We integrate all three approaches into our cache management scheme and derive cost models to calculate cache miss penalty for each type of miss.

4.1 RPDR: Remote Partial Data Recovery

4.1.1 Methodology

We propose *Remote Partial Data Recovery (RPDR)*, which "patches" a partially unavailable dataset by fetching missing data portions from a remote primary copy in an on-demand fashion. This approach takes advantage of the following two trends. First, scientific datasets cached locally tend to have remote primary copies stored at systems such as mass storage and data centers with relatively high availability. Second, bulk scientific data movement tools such as HSI, GridFTP [3, 4], and LoRS [39] are increasingly supporting "extended retrieve" features. Of particular interest to us is their ability to fetch partial copies of a dataset, which enables the retrieval of a section of the dataset, defined by its starting offset and extent.

Note that RPDR is designed to work in conjunction with local caching of scientific datasets. Therefore the I/O and network workload generated by recovery will not exceed those of the original data access pattern without local caching. Meanwhile, we have shown that the negative performance impact of data fetching/serving on donated storage is modest [57] and can be well controlled [52].

RPDR augments the basic striping-aware cache replacement policy with mechanisms to handle full or partial data misses in a unified manner. This is achieved as follows. When a dataset D is accessed, the manager node searches for the dataset in the cache. If D is found, the manager checks the current status of storage nodes that host this dataset. For each contiguous segment of D striped in a round-robin fashion on the same set of storage nodes $N_1, N_2, ..., N_k$ (where $k \leq w$ and w is the target stripe width), the manager identifies the length of the segment l (each node hosts l chunks) and the number of unavailable nodes f ($f \leq k$). If $f > 0$, $f \times l$ data chunks have to be recovered. Before data can be recovered, sufficient space needs to be created through eviction. Cache replacement is invoked to evict $f \times l$ chunks, followed by the invocation of the striping algorithm to reallocate this space to the data being fetched from D's primary copy at an external source.

This reallocation is accomplished by first sorting the storage nodes on available space, then performing the greedy striping process for the missing chunks of D with an initial stripe width of f. This way, the number of storage nodes communicating with the client simultaneously is controlled to be close to the target stripe width, w. Chunks of D on the f unavailable storage nodes are marked by the manager as dirty. If these nodes become available again, they will synchronize the status of their local data chunks during their first interaction with

the manager. At that point, these chunks become free and may be allocated to other datasets.

Once sufficient room has been made for the missing chunks of *D*, RPDR retrieves these chunks from the dataset's primary storage location (typically at a remote site such as an archival system) to "patch" the cached copy of *D*. FreeLoader assumes that a remotely-accessible dataset has a web address or URI, which is used as a unique identifier for the dataset in the aggregate storage namespace. The protocol and the command used to download the dataset are also managed by FreeLoader as a part of the dataset metadata.

It can be seen that with this policy, a full miss is treated as a special case of partial misses, only with additional operations such as registering the dataset and creating metadata entries. In this case, eviction has to be performed to accommodate the entire dataset, *D*, and the greedy striping starts with the target stripe width. Tail misses are handled in a similar fashion, although the dataset entry is already in the cache.

With RPDR, a partially cached dataset can still be accessed while being patched, therefore we adopt a modified cache management policy from the one defined in Section 3. Eviction is performed at the chunk level rather than at the dataset level. Thus, instead of evicting chunks from only one dataset at a time, a round-robin method is used to evict tail chunks from datasets with the same heat index. In doing so, we exploit the fact that a tail patch operation can be easily overlapped with client-serving of the cached chunks.

4.1.2 Coupling RPDR with Prefix Caching and Collective Download

Remote partial data recovery can also be coupled with a combination of two techniques that we applied previously to FreeLoader [33], namely *prefix caching* and *collective download*, for more efficient—and more sophisticated—cache management.

As mentioned earlier, RPDR uses remote accesses to patch disjoint "holes" or a suffix in a dataset, caused by node unavailability, cold miss, or partial data eviction. One obvious approach is to have multiple storage nodes download their target stripes directly from the remote source. With data striping (where stripe sizes typically range from several hundred KBs to dozens of MBs), however, this would result in numerous small requests. A more efficient alternative is to let each node download a large, contiguous data segment and perform local data shuffling to exchange data stripes. We refer to this as *collective download*, following similar concepts of *collective I/O* [8, 28, 35, 48, 54]. The downloaded data is simultaneously 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.

To leverage the partial data transfer capability of popular scientific data migration tools such as GridFTP [3, 5] and HSI [22], we used Expect [31], a tool specifically geared towards automating interactive applications. We have 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. More design and implementation details can be found in our previous publication [33].

Next, we couple collective download and prefix caching together with RPDR for better space utilization. With prefix caching, only a prefix of the dataset is cached on initial import. It allows us to "bootstrap" the data download process with the in-cache prefix, as a lazy method in the sense that complete retrievals are not initiated until datasets are actually accessed. Upon the access request arrival, the cached prefix is served to the client while the missing suffix is fetched and patched transparently from the primary data source. Even then, the retrieved suffix is stored in, but streamed through the cache. By overlapping in-cache data access with remote data retrieval, 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 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, tape system file loading, etc.), and the size-dependent cost of parallel data transfer at the aggregate rate $R_{collective}$. We assume that $R_{collective} < R_{client}$, which means the client fetches data from the local cache faster than directly from the external data source—implying the need for a cache. The equation below equates client access time and the time of the two components of patching,

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

Solving for the prefix size, we get

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

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 $R_{collective}$ 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.

Note that after the re-access, the patched tail is discarded after the re-access. When the cache is full and more space is needed for a new dataset, chunks are evicted from the tail of the cached prefix just like in the base version of RPDR. This results in a tail miss with a visible performance penalty while accessing that particular dataset.

In Section 5.3, we will evaluate both the base RPDR strategy and the enhanced version with prefix caching and collective download.

4.2 Striping-enabled Local Recovery through Data Redundancy

A storage aggregation system built on an unreliable distributed storage fabrics can also improve data availability through local data recovery, typically by storing redundant data. We examine two commonly used local recovery mechanisms, *replication* and *erasure coding*, in the distributed cache setting and compare them with our proposed strategies.

Replication is widely used in parallel or distributed file systems [21, 45] and storage aggregation systems [2, 9, 17]. With replication, a dataset is duplicated a number of times in a storage system, to prepare for disk/network failure. The advantage of replication is its simplicity and ability to achieve high data availability. For example, a study has shown that a 99.99% data availability can be obtained by replicating all files three times in a distributed file system built on top of unreliable donated disk space [9]. This study was based on the same Microsoft node availability trace used in this paper (see Section 5.1 for more details). The apparent disadvantage of replication is its space overhead. While this is an acceptable overhead for distributed file systems, a cache built from space contributions works differently. The cache space is a finite amount of storage based on desktop space donations. For a distributed cache storing "hot" data items, replicating each dataset n times basically cuts the cache space to $1/n$ of its original size. This results in further cache misses, which is another form of data unavailability and may cause more performance penalty in data accesses

compared to partial dataset loss due to temporal node unavailability.

Erasure coding [42, 43] is a general scheme for error correction and data recovery, first proposed for network communication and more recently used in peer-to-peer systems and distributed storage systems. For each n blocks of the original data, an erasure code can encode m additional *redundant blocks*, allowing the original data to be reconstructed from any n out of the $n+m$ blocks. Obviously, with a moderate m, erasure codes can deliver comparable or better data availability than replication [59], while consuming a fraction of extra space as required by the latter. This makes erasure coding more appealing in data caches built out of aggregated space. However, the major concern with it is the extra encoding and decoding overhead. In a storage aggregation system, space donors may hesitate to donate any more resource than they already do. Thus, any computationally intensive encoding or decoding will need be performed at the client, which imports or subsequently accesses a dataset.

To conduct performance comparisons between the different data recovery approaches, we need to first determine the replication and erasure coding strategies in a data-striping environment too, as briefly outlined below.

For replication-k, where k is the degree of replication, we perform striping of k copies of each dataset using our greedy striping algorithm defined in Section 3. If the cache is too small to fit in k replicas, we create as many replicas as allowed by the cache size. If, during a subsequent access, we find there is additional space to create more replica(s) for the dataset being accessed (this is possible due to the unreliable nature of the storage nodes, the total *available* cache size may vary), such replica(s) will be stored. The k replicas are managed individually by the chosen cache replacement policy. The access-history-based heat index of each dataset is adjusted by a factor of the number of its replicas, so that dataset with more replicas are more likely to be evicted in cache replacement. At dataset retrieval time, data chunks can be selected from multiple replicas. If there are chunks missing from all in-cache replicas due to node failures and/or tail eviction, the access is considered a miss.

For erasure coding, we encode the w (where w is the stripe width) data chunks into m redundant chunks, and stripe the dataset with a stripe width of $(w+m)$. Extra care is taken to make sure that the stripe width does not "shrink", so that each round of chunks actually occupy $(w+m)$ different nodes. Note this may call for additional cache eviction in search of enough nodes. Therefore, the space overhead of this method will end up being larger than m/w, since excessive amounts of data chunks may be evicted, rendering a smaller effective cache space.

4.3 Data Recovery Cost Model

To evaluate the performance of the three recovery approaches, we need to model the cost of data recovery for the various types of cache misses. Since most scientific data processing applications access data in sequential or approximately sequential styles, we model the client's data access as a sequential scan to retrieve a dataset's chunks in order.

First, we analyze the data recovery cost for RPDR. In this case, the storage nodes involved in importing or patching a dataset D will download data chunks in parallel, sending these chunks to the client while saving them to their local disks. Since the manager can perform cache replacement and decide in very short time which storage nodes need to fetch missing data from a remote site, we allow all these nodes to start downloading together, while the client requests data chunks from the nodes according to the chunk map. Since different rounds of striping may involve a different subset of nodes, "holes" in a partially available dataset may start or end at arbitrary positions.

Let R_n be the download throughput of each patching node, when n such nodes are retrieving interleaved data chunks simultaneously. Since we select the target stripe width, w, to be large enough to saturate the client's incoming network connection, the *expected data serving throughput* for each storage node is $R_e = \frac{R_C}{w}$, where R_C is the maximum client network read rate. The question is whether the download throughput at the patching nodes can catch up with R_e. When $R_n \geq R_e$, the remote data retrieval at a patching node is faster than the rate that node is supposed to deliver data

Table 1 Cost of data accesses with different recovery methods

Dataset's caching status	Recovery method		
	RPDR	REPL	EC
Hit	T_{local}	T_{local}	T_{local}
Partial miss	T_{patch}	$min(T_{tail-patch}, T_{client})$	$max(T_{local}, T_{decode})$
Tail miss	$max(T_{local}, T_{tail-patch})$	$min(T_{tail-patch}, T_{client})$	$max(min(T_{tail-patch}, T_{client}), T_{encode})$
Total miss	$min(T_{tail-patch}, T_{client})$	$min(T_{tail-patch}, T_{client})$	$max(min(T_{tail-patch}, T_{client}), T_{encode})$

to the client, partial data recovery does not have a client-visible performance impact. Otherwise, a cache miss penalty may be observed.

With the chunk map, for each patching node P_i, we can compute s_i, the number of chunks that P_i is responsible for fetching remotely, as well as the expected point in time $T_{e,i,j}$ when the jth missing chunk from P_i needs to be delivered at the client. By aligning the missing chunks together from all patching nodes, we compute the point in time $T_{p,i,j}$ when the jth missing chunk is retrieved from the remote primary copy:

$$T_{p,i,j} = \sum_{k=1}^{j} \frac{chunk_size}{R_{n_k}},$$

where n_k is the number of patching nodes fetching data concurrently when P_i is fetching its kth chunk. Then the cache miss penalty, $T_{penalty}$, is calculated as

$$\max_{i,j} (T_{p,i,j} - T_{e,i,j}),$$

the maximum latency of data patching among patching nodes. The total access cost, T_{patch}, is then defined as $(T_{local} + T_{penalty})$.

The above RPDR recovery scheme and corresponding cost computation are based on observations collected from parallel partial data recovery benchmarking experiments. From our experiments we have found that most data sources show good download bandwidth scalability when we increase the number of patching nodes, each fetching noncontiguous (or contiguous when performing collective download) data chunks. Hence, we let all patching nodes start downloading their data chunks at the beginning of a client data access. This will allow the missing data to be fetched early, without affecting the rate that "urgent" data chunks are downloaded and served to the client. More details about the design and results of related benchmarking tests will be discussed in Section 5.2.

A tail miss—or a total miss, for that matter—is conceptually same as a partial miss. However, with collective download, they can be implemented in a much more efficient way. As discussed in Section 4.1.2, for a large missing segment in a dataset, it is efficient to have multiple nodes downloading contiguous sections of that segment, while also performing local data exchange to achieve a desired striping pattern. Suppose the collective download (including the shuffling process) can be performed at a rate of $R_{collective}$, the tail patching cost $T_{tail-patch}$ is calculated as $size(D)/R_{collective}$.

Finally, the above total miss cost model may need to be adjusted depending on the client's direct access rate from the target external source. With certain data sources, a direct download of the entire dataset by the client may yield better aggregate access rates. With the benchmarking results, it is easy to determine whether the client download rate R_{client} is indeed higher than the aggregate patching rate for a total miss wR_w (where w is the stripe width). If this is the case, we calculate the cost of a total miss as the duration of a client download, which can be overlapped with striping data chunks to storage nodes.[1]

The recovery cost for the other two approaches, replication and erasure coding, is much more straight-forward. Table 1 summarizes how we characterize the cost of accessing a dataset in an unreliable distributed cache with data striping,

[1]This did not happen with the four external data sources that we benchmarked.

for the three data recovery methods we study: remote partial data recovery (RPDR), replication (REPL), and erasure coding (EC). These models are used in our trace-driven simulations to calculate the total access time for a sequence of accesses. Note that different methods will generate different distributions of accesses, falling into the cache status categories.

When the dataset being accessed is in-cache and all its chunks are available, the cost of access is the same across all three methods: the local data transfer time between the storage nodes and the client (T_{local}).

When the dataset is a total miss, with RPDR or REPL the access cost is the downloading time to bring the missing dataset into the cache. This is done either through the storage nodes ($T_{tail-patch}$) or directly between the external data source or the client (T_{client}), depending on which one is faster. Again we assume that the cost of striping data to the appropriate storage nodes can be hidden by the downloading time. In calculating REPL's total miss cost, we do not count the dataset size k times, considering that the replication can occur between the storage nodes themselves without involving the client. For EC, although the encoding has to be performed at the client, if the client downloading rate is very low, data can still be downloaded in parallel by the storage nodes and streamed into the client for encoding. The three activities (parallel downloading, client encoding, and striping the encoded data) can again be pipelined. Hence the total miss cost is the larger of the downloading time ($min(T_{tail-patch}, T_{client})$) and the encoding time (T_{encode}).

A tail miss is treated as a total miss for REPL and EC. But for RPDR, since the tail patching can be hidden behind the prefix access, the total access time is the larger of T_{local} and $T_{tail-patch}$.

Finally, when there is a partial data miss, with RPDR the access cost is T_{patch}. With REPL, there is essentially no partial miss: if a data chunk is missing from all replicas, the entire dataset is treated as a total miss. With EC that encodes n original data chunks into m redundant chunks, if data needs to and can be reconstructed, the partial miss cost is the cost of the slower process between client data retrieval and decoding, assuming the two can be perfectly overlapped.

5 Performance Results

5.1 Simulation Overview

We perform trace-driven simulations to evaluate our proposed striping-enabled cache management and partial data recovery strategies.

Our simulator takes as input two traces: a *node availability trace* and a *dataset access trace*. The node availability trace logs the status of a group of nodes within an institution over a period of time and reflects the availability of storage nodes participating in a storage aggregation system. The dataset access trace logs external scientific dataset access information (such as file name, size, access time, client host name, etc.) within an institution and reflects the distributed scientific data cache users' access pattern. Our simulator combines these two traces to examine data availability, cache behavior, and recovery performance in a realistic setting.

We used the desktop availability trace from Microsoft that consists of "up/down" measurements of 50,000 desktop workstations in their campus, for a period of 840 h [2]. This trace (called MS trace for brevity) was collected by recording the results of periodic "ping" operations to each workstation: a machine is marked as either "up" or "down" for each of the 840 h. Note that this requires the coarsening of access time in processing the dataset access trace. *I.e.*, while we preserve the original order and access times of dataset access trace entries, a relevant storage node is considered available or unavailable according to the availability trace for *each entire hour*.

In order to verify that the collective node behavior portrayed by the MS trace is also reflective of academic settings, we performed a similar data collection experiment using 600+ workstations at the Department of Computer Science in North Carolina State University. The experiments were conducted for a duration of over 4 months. Both traces show similar availability trends, with over 60% of the nodes available for over 90% of the time.

In using the MS trace, we performed multiple tests, each of which used a group of randomly selected 50 nodes from the node pool (50,000) as storage nodes participating in the aggregate cache.

Since this paper does not focus on heterogeneity issues, we assume similar I/O rates, network capabilities, and space contributions across the storage nodes.

For dataset accesses, we obtained two real-world traces, Jasmine and ARM. Jasmine contains the Jefferson Laboratory researchers' access logs to the high-energy physics data hosted at the Jefferson Lab Asynchronous Storage Manager (JASMine). ARM contains the Oak Ridge National Laboratory (ORNL) researchers' access logs to the Atmospheric Radiation Measurement (ARM) archive hosted at the HPSS mass storage system at ORNL. Table 2 summarizes statistics about the two access traces. One might argue that the dataset sizes in these traces do not look large enough to exceed a workstation's local capacity. However, scientific users often access collections of datasets [36], each of which is on the order of a few GBs, for better portability among file systems. In addition, as we pointed out earlier, accessing data in a striped fashion from a distributed cache has a performance advantage, which justifies the use of this cache for these traces.

To reduce the simulation time, we filtered the original Jasmine trace to retain only accesses to larger datasets (those above a size threshold of 2000MB). Table 2 shows the number of entries in the filtered trace (the original trace contains 644,132 entries). We consider the trimming reasonable since scientists would be more interested in using a distributed cache for their large datasets.

5.2 Parameterizing the Simulator

This section presents performance parameters (such as R_e and R_n as described in Section 4.3 and the client download rate R_{client}) obtained from our aggregate storage prototype using multiple external data sources. These results are used in the access time calculation in our simulation.

Table 2 Data access trace statistics

Trace	ARM	Jasmine
Duration	157.5 days	19.1 days
No. entries	3709	4000
No. unique datasets	382	1686
Average dataset size	253.7 MB	2047.0 MB

5.2.1 Benchmarking Testbed Configuration

Our testbed (Fig. 2) depicts a scientist's local research environment, where his/her powerful client machine has access to external data sources such as parallel/archival file systems and Web data repositories, using various data movement tools.

We installed the FreeLoader aggregated storage cache in this setting to study its ability to transparently patch datasets from external data sources. Our testbed spreads across Oak Ridge National Laboratory (ORNL), North Carolina State University (NCSU), and the TeraGrid (a Nationally deployed Grid infrastructure). It comprises 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): (i) the HPSS [13] archival storage system (HSI) at ORNL, accessed through the Hierarchical Storage Interface [22] client software, (ii) a web repository at NCSU (wget-R) accessed through the wget interface, and (iii) two GridFTP servers, enabling access to parallel file systems on the TeraGrid sites, ORNL (GridFTP-L) and PSC (GridFTP-R). We used the UberFTP client interface to access the GridFTP servers. Figure 2 also shows the connectivity of these

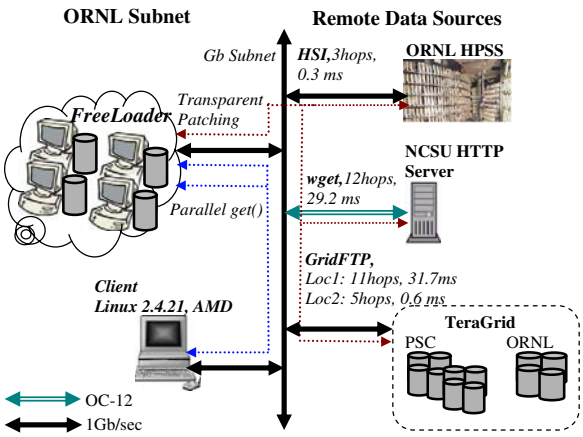


Fig. 2 Benchmarking testbed

remote data sources to the ORNL subnet, which results in varied patch bandwidth.

2. The aggregate storage cache at ORNL: an aggregate storage of 0.5TB with 15 storage nodes (donating 7-60GBs each) and one manager. Donors have dual Pentium III, Linux 2.4.20-8 kernel, and 100Mb/sec or 1Gb/sec Ethernet. The storage nodes are equipped to patch from the remote sources using appropriate protocols as well.
3. A client machine at ORNL with Dual AMD Opteron, Linux 2.4.21 and GigE, running the aggregate storage client component. It is at most five hops away from any of the storage nodes in the aggregate storage cloud.

5.2.2 Benchmarking Results

In order to obtain the downloading rate of the entire dataset from the external source, (R_{client}), client access rate of in-cache data (R_{C_w}, where w is the stripe width), and partial data recovery rate (R_n, where n is the number of nodes concurrently patching), we conducted a series of tests based on our prototype implementation of the aggregate storage framework. Our bandwidth benchmarking and simulation tests used a target stripe width of 10, since we observed that for our testbed, the client access rate saturated at that width (see Fig. 1b).

To study recovery, we mimic node failures and have substitute nodes patch, from the external source, the "holes" due to the loss of stripes. When n nodes are patching concurrently, each is retrieving one stripe, chunk by chunk. The total amount of data retrieved is then $n/10$ of the entire dataset's size. Therefore, when $n = 10$, the whole dataset is downloaded in parallel. Through our tests, we have found that for the data sources and protocols used, the patching rate per node R_n does not vary much as we increase n, the number of nodes patching concurrently. The decrease in bandwidth ranges between 0 (GridFTP-L) and 4.7% (wget-L) when n grows from 1 to 10. This is most likely due to the strided data access pattern. Therefore, we simplify the partial data recovery model in Section 4.3 and substitute R_n with a single R parameter, which is conservatively chosen as R_{10}.

Similarly, we examined whether the per-node patching rate could be affected by the patching node's concurrent data serving to the client, since the network interface is shared. Again, we found the impact to be very small, ranging from 0 (GridFTP-L) to 10% (wget-L). We select the lower R number, pessimistically, assuming the patching activity fully overlaps with serving data to the client.

In addition, we measure the collective downloading rate for patching a tail miss and importing entire datasets, when 10 nodes are fetching a contiguous region of a dataset. The rate is calculated by including the cost of both the parallel downloading and the inter-node data shuffling activities. We further measure the rate that an entire 1GB dataset is downloaded by the GigE client, R_{client}, as an alternative to fetching the data in case of a total miss.

Table 3 summarizes our results. The results reveal two facts. First, the downloading rates are significantly lower than allowed by the storage nodes' network interface. In the case of protocols like GridFTP, which are optimized for large transfers, the overhead of multiple small accesses, including authentication and connection establishment each time around, results in an overall low patch rate despite high-speed networks. This suggests that we can minimize the cost of patching by performing large remote I/O and local data reorganization. With the wget tests, however, latency incurs a major cost. Second, as mentioned above, with the scalability we observed, having 10 storage nodes collectively download the dataset and exchange stripes (at $10 \times R_{collective}$) is faster than having the GigE client download the whole dataset (at R_{client}). This appears to be true for all data sources.

To get a rough estimate of R_e, the expected chunk delivery rate from each storage node to the

Table 3 Throughput parameters benchmarked, in MB/s, using a 1GB dataset

Data source	wget-R	GridFTP-L	GridFTP-R	HSI
R_{client}	2.3	20.5	3.5	11.5
R	1.1	3.8	1.41	0.91
$R_{collective}$	2.0	7.5	2.02	1.3

All numbers are per-node rates

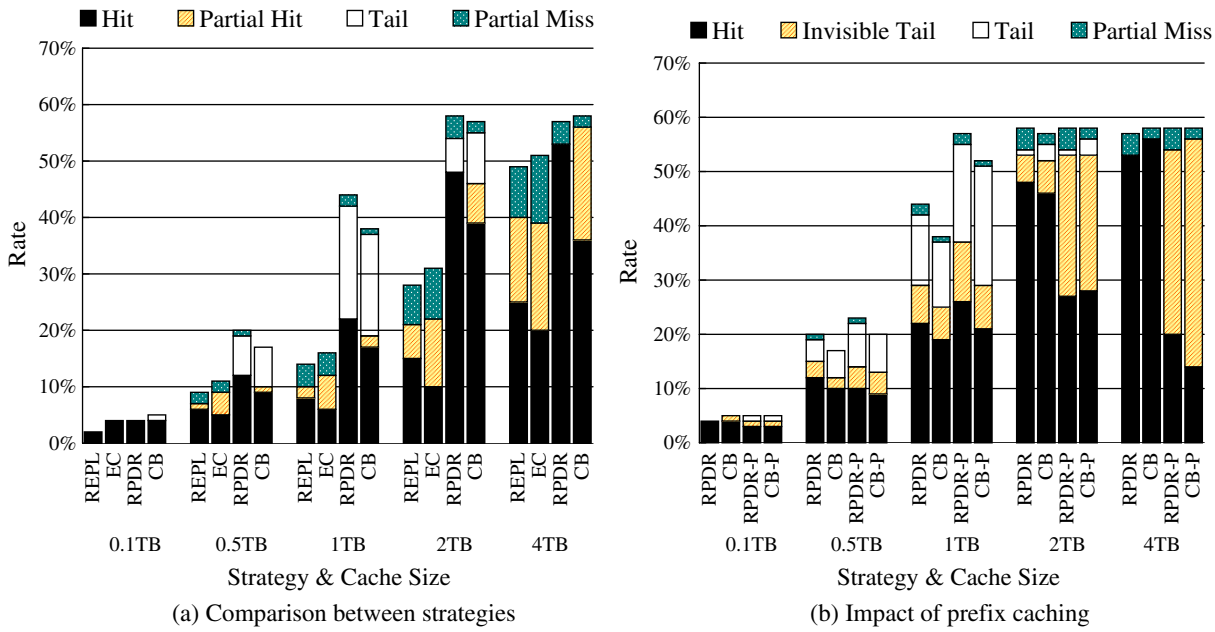


Fig. 3 Jasmine trace hit/miss rates (**a**, **b**)

client, we divided the FreeLoader client access rate (89MB/s for a stripe width of 10 for a 1GB dataset) by the stripe width.

Finally, to estimate the encoding/decoding cost for erasure coding, we benchmarked the parity check throughput using an efficient implementation by James Plank (http://www.cs.utk.edu/~plank/plank/gflib/parity_test.c). The parity check throughputs we measured on multiple workstations are all higher than 100MB/s, and hence will not be a performance bottleneck since the encoding/decoding cost will be hidden by the data transfer between the client and the storage nodes.

5.3 Simulation Results

5.3.1 Jasmine Trace Results

First, we examine the cache behavior with data striped on unreliable nodes. The stripe width used is 10.[2] All experiments were repeated 5 times. Since the variance in the results were reasonably small (<10%), we show the average numbers without error bars. A uniform space donation is used across the 50 nodes.

Figure 3a shows the caching performance using the Jasmine trace. We varied the total cache size from 100GB to 4000GB. For each cache size, the performance of four strategies is shown side by side: REPL, EC, RPDR, and CB. CB stands for "combination", where we use RPDR as a backup for EC. In other words, a partial data miss that cannot be repaired by local redundancy using EC, is patched using RPDR. In this case, both RPDR and CB use the base version, without prefix caching. For each test, we measure the percentage of accesses (by size) in each hit/miss category: hit, partial hit, tail miss, and partial miss. The rest are total misses. "Partial hit" denotes a miss that can be recovered locally through REPL or EC. With replication or the fast EC decoding we measured, there is no performance penalty. "Partial miss", on the other hand, denotes a miss that requires remote accesses. For RPDR, this means that there are holes missing in the requested dataset. For REPL, EC, and CB, this means that the cached dataset cannot be completely recovered with local redundant chunks.

[2]In our experiments we have varied the stripe width and the results across different widths did not show significant difference. Therefore we omit the other stripe widths due to space limitation.

As mentioned earlier, a tail miss with RPDR or CB is patched using collective download and the patch operation can be easily overlapped with the in-cache access of the dataset. Hence, for these two strategies, we categorize the cached part of those datasets as a "hit" and the missing part as a "tail miss". This does not apply to REPL or EC because there any dataset that cannot not be recovered locally is a total miss.

As expected, for all four strategies, both the hit rate and the partial hit rate increases as the cache size grows. The partial miss rate also increases due to the fact that more datasets are cached. The tail miss bar is visible only for RPDR and CB since these two strategies allow partial data recovery and cache eviction may select the tails of multiple datasets simultaneously. This rate first grows then shrinks as the cache gets larger, because the cache saturates before reaching 4TB and the reduced space shortage causes fewer tail misses.

Figure 3a indicates that RPDR and CB clearly outperform REPL and EC. RPDR has significantly higher hit rate than the combined hit/partial hit rates achieved by adding local data redundancy. CB has a lower hit rate than RPDR due to its space overhead, but compensates that with its partial hit rate due to erasure coding. This result indicates that by fetching data from external sources, RPDR is able to achieve better cache space utilization than REPL and EC. Meanwhile, by allowing data patching from external sources as a fall-back approach when local data redundancy is inadequate, CB appears to be equally appealing. CB incurs the same space overhead as EC. However, reconstructing a partially cached dataset that was otherwise not recoverable through decoding makes it more space-efficient.

Although not shown in the chart, we found that most of the partial data unavailability caused by node failure involves a single failing storage node. Therefore, by storing only two replicas, REPL results in a small partial miss rate (which will be treated as a total miss in calculating the miss penalty). EC has more partial miss, as a result of being able to cache more datasets. Overall, EC's caching performance is better than REPL considering both the "hit" and "partial hit" categories.

Figure 3b compares different versions of RPDR and CB, with and without the prefix caching enhancement. The versions with prefix caching are denoted as "RPDR-P" and "CB-P", respectively. With prefix caching, we only cache a prefix of each dataset. The prefix size is determined by calculating the maximum length of the missing tail such that the patching cost can be hidden by the prefix access time. When more cache space is needed, however, chunks from the cached prefix will be evicted, causing a remote patching penalty. Therefore, we divide the tail miss data into two parts: "invisible tail", whose recovery can be hidden by the prefix access, and "tail", whose fetching cost is visible to the client.

As can be seen from Fig. 3b, prefix caching significantly increases the total amount of tail miss ("invisible tail" plus "tail"). However, the combined size of cache hit and invisible tail miss is greater compared to when not using prefix caching. This is true for both RPDR and CB. The improvement is more obvious for medium cache sizes, 0.5TB and 1TB, where a considerable portion of hot datasets is cached but the total cache space is still insufficient. The base version (without prefix caching) also evicts chunks from the tail of victim datasets. With prefix caching, however, the initial, in-cache prefix is decided by the size of each individual dataset and the remote recovery cost. Therefore, a larger portion of the access falls into the "invisible tail" category. This trend becomes more obvious when the cache size is larger, where there is fewer visible tail miss and many invisible ones.

Figure 4 depicts, for each external data source, the total access time for the Jasmine trace calculated from the above tests using our cost models. For each cache size, the total access time is computed by adding the access time of each trace entry. Since Fig. 3b shows that the versions with prefix caching work better than the base versions of RPDR and CB, we only plot the performance of RPDR-P and CB-P.

Our results show that once again CB and RPDR clearly outperform REPL and EC in most cases. The only exception is with small to medium cache size with our fastest source (GridFTP-L). In this case, the collective download rate is high

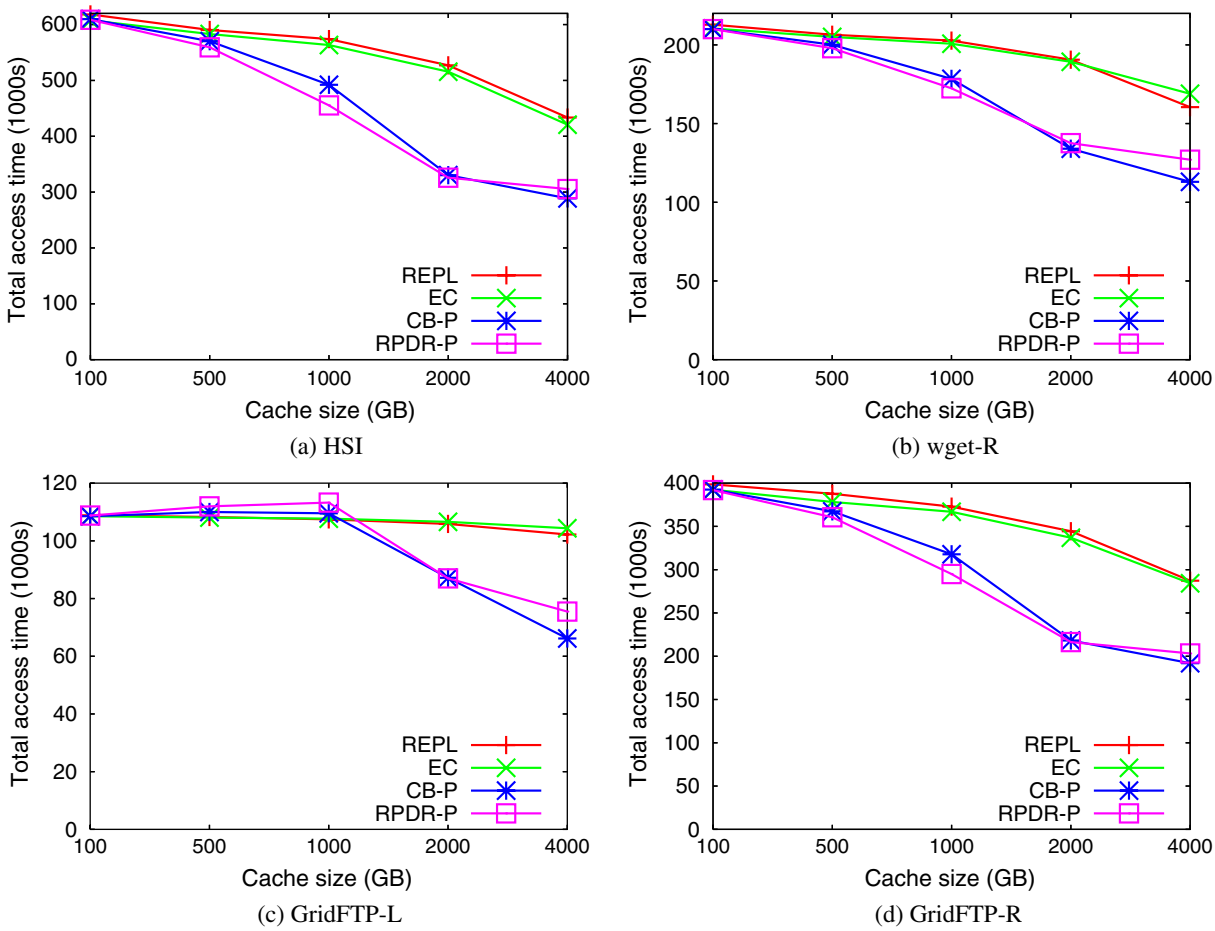


Fig. 4 Jasmine trace access times (**a**–**d**)

(7.5 MB/s per node, making 75MB/s when 10 nodes are concurrently downloading). This way, the higher rate of a total miss caused by REPL and EC becomes less expensive, while the "hole patching" of RPDR and CB appears relatively slow. In all other cases, RPDR-P and CB-P outperform approaches using local data redundancy, especially with medium to large cache sizes. RPDR-P outperforms REPL by up to 37% and EC by up to 36%, while CB-P outperforms REPL by up to 38% and EC by up to 36%.

Despite their large difference in space usage, EC and REPL have very similar performance. In most cases EC performs slightly better, which is consistent with the caching performance result (Fig 3a). Meanwhile, neither of RPDR-P and CB-P appears to be a clear winner and they have very similar performance.

5.3.2 ARM Trace

Due to space limitations, we present just the access time results for the ARM trace. Since this trace is significantly smaller compared to the Jasmine trace in the total dataset size, the cache sizes that we used are also much smaller. Also, due to the smaller dataset sizes, we use the base versions of RPDR and CB rather than those with prefix caching.

Figure 5 portrays the access time results with the ARM trace for the four external data sources. Here, CB appears to be a consistent good choice.

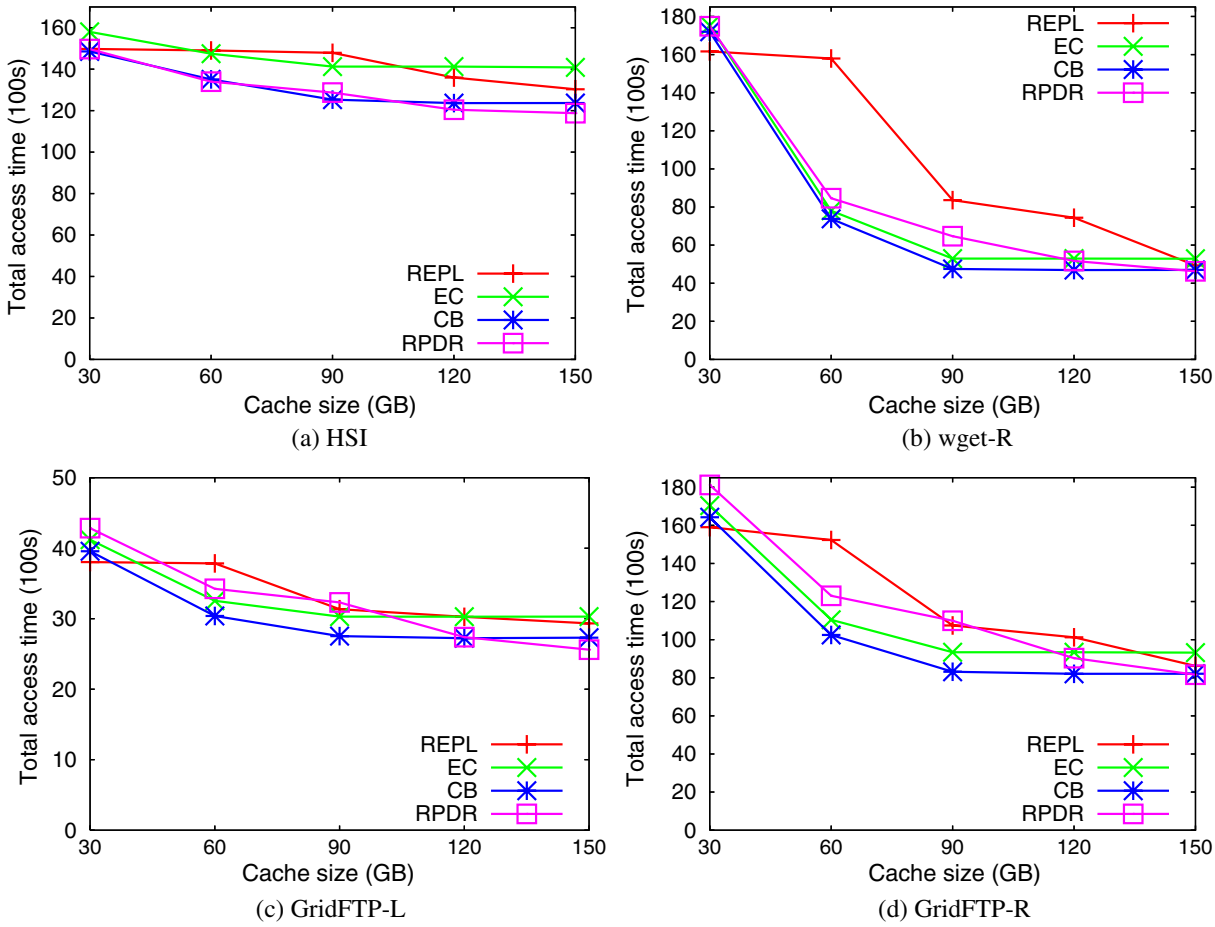


Fig. 5 ARM trace access times (**a**–**d**)

In this set of experiments, CB outperforms REPL by up to 54% and EC by up to 12.2%.

6 Conclusion

In this paper we studied data availability and recovery issues with distributed caches built on unreliable, donated desktop space. We proposed a striping-aware cache management strategy for this unique environment, upon which we studied three data recovery approaches (remote partial data recovery, replication, and erasure coding).

We demonstrated, with real-world scientific data access traces and performance parameters measured from a distributed cache prototype, that our proposed approaches based on direct data recovery from remote data sources perform significantly better than the ones totally relying on the local data redundancy. In particular, the CB approach, which combines erasure coding with remote partial data recovery, delivers the best performance in the majority of test cases.

References

1. Acharya, S., Smith, B.: 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)
2. Adya, A., Bolosky, W., Castro, M., Chaiken, R., Cermak, G., Douceur, J., Howell, J., Lorch, J., Theimer, M., Wattenhofer, R.: FARSITE: federated, available, and reliable storage for an incompletely trusted environment. In: Proceedings of the 5th

Symposium on Operating Systems Design and Implementation (2002)
3. Allcock, W., Bresnahan, J., Kettimuthu, R., Link, M.: The globus striped GridFTP framework and server. In: Proceedings of Supercomputing '05 (2001)
4. Bester, J., Foster, I., Kesselman, C., Tedesco, J., Tuecke, S.: GASS: a data movement and access service for wide area computing systems. In: Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems (1999)
5. Bester, J., Foster, I., Kesselman, C., Tedesco, J., Tuecke, S.: Gass: a data movement and access service for wide area computing systems. In: Proceedings of the 6th Workshop on Input/Output in Parallel and Distributed Systems (1999)
6. Blake, C., Rodrigues, R.: High availability, scalable storage, dynamic peer networks: pick two. In: Proceedings the 9th Workshop on Hot Topics in Operating Systems (HotOS) (2003)
7. Bolosky, W., Douceur, J., Ely, D., Theimer, M.: Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In: Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (2000)
8. Bordawekar, R., Rosario, J., Choudhary, A.: Design and evaluation of primitives for parallel I/O. In: Proceedings of Supercomputing '93 (1993)
9. Butt, A., Johnson, T., Zheng, Y., Hu, Y.: Kosha: a peer-to-peer enhancement for the network file system. In: Proceedings of Supercomputing (2004)
10. Byers, J., Luby, M., Mitzenmacher, M., Rege, A.: A digital fountain approach to reliable distribution of bulk data. In: Proceedings of the ACM SIGCOMM Conference (1998)
11. Cluster File Systems, Inc. Lustre: a scalable, high-performance file system. http://www.lustre.org/docs/whitepaper.pdf (2002)
12. Cohen, E., Shenker, S.: Replication strategies in unstructured peer-to-peer networks. In: Proceedings of the ACM SIGCOMM Conference (2002)
13. Coyne, R.A., Watson, R.W.: The parallel I/O architecture of the high-performance storage system (hpss). In: Proceedings of the IEEE MSS Symposium (1995)
14. Dahlin, M., Wang, R., Anderson, T., Patterson, D.: Cooperative caching: using remote client memory to improve file system performance. In: Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (1994)
15. Douceur, J., Bolosky, W.: A large-scale study of file-system contents. In: Proceedings of SIGMETRICS (1999)
16. Druschel, P., Rowstron, A.: Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In: Proceedings of the 18th ACM Symposium on Operating System Principles (2001)
17. Kubiatowicz, J., et al.: Oceanstore: an architecture for global-scale persistent storage. In: The 9th International Conference on Architectural Support for Programming Languages and Operating Systems (2000)
18. Feeley, M., Morgan, W., Pighin, F., Karlin, A., Levy, H., Thekkath, C.: Implementing global memory management in a workstation cluster. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles (1995)
19. Forney, B., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: Storage-aware caching: revisiting caching for heterogeneous storage systems. In: Proceedings of the USENIX Conference on File and Storage Technologies (FAST) (2002)
20. Gadde, S., Chase, J., Rabinovich, M.: A taste of crispy squid. In: Proceedings of the Workshop on Internet Server Performance (1998)
21. Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. In: Proceedings of the 19th Symposium on Operating Systems Principles (2003)
22. Gleicher, M.: HSI: hierarchical storage interface for HPSS. http://www.hpss-collaboration.org/hpss/HSI/ (2005)
23. Gray, J., Liu, D., Nieto-Santisteban, M., Szalay, A., Heber, G., DeWitt, D.: Scientific data management in the coming decade. Technical Report MSR-TR-2005-10, Microsoft (2005)
24. Gruber, S., Rexford, J., Basso, A.: Protocol considerations for a prefix-caching proxy for multimedia streams. http://www9.org/w9cdrom/349/349.html (2009)
25. Iamnitchi, A., Ripeanu, M., Foster, I.: Small-world file-sharing communities. In: Infocom (2004)
26. Jin, S., Bestavros, A., Iyengar, A.: Network-aware partial caching for internet streaming media delivery. ACM/Springer Multimedia Syst. **9**(4) (2003)
27. Kallahalla, M., Varman, P.J.: PC-OPT: optimal offline prefetching and caching for parallel I/O systems. IEEE Trans. Comput. **51**(11) (2002)
28. Kotz, D.: Disk-directed I/O for MIMD multiprocessors. In: Proceedings of the Symposium on Operating Systems Design and Implementation (1994)
29. Lee, J., Ma, X., Ross, R., Thakur, R., Winslett, M.: RFS: efficient and flexible remote file access for MPI-IO. In: Proceedings of the IEEE International Conference on Cluster Computing (2004)
30. Lee, J., Ma, X., Winslett, M., Yu, S.: 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)
31. Libes, D.: The expect home page. http://expect.nist.gov/ (2006)
32. Litzkow, M., Livny, M., Mutka, M.: Condor—a hunter of idle workstations. In: Proceedings of the 8th International Conference on Distributed Computing Systems (1988)
33. Ma, X., Vazhkudai, S., Freeh, V., Simon, T., Yang, T., Scott, S.L.: Coupling prefix caching and collective downloads for remote data access. In: Proceedings of the ACM International Conference on Supercomputing (2006)
34. National Center for Biotechnology Information. Genbank overview. http://www.ncbi.nlm.nih.gov/Genbank/GenbankOverview.html (2004)

35. Nieuwejaar, N., Kotz, D.: The Galley parallel file system. Parallel Comput. **23**(4), 447–476 (1997)
36. Otoo, E.J., Rotem, D., Romosan, A.: Optimal file-bundle caching algorithms for data-Grids. In: Proceedings of Supercomputing (2004)
37. Padmanabhan, V.: Using predictive prefetching to improve World Wide Web latency. In: Proceedings of ACM SIGCOMM (1996)
38. Patterson, D., Gibson, G., Katz, R.: A case for redundant arrays of inexpensive disks (RAID). In: Proceedings of the ACM SIGMOD Conference (1988)
39. Plank, J., Atchley, S., Ding, Y., Beck, M.: Algorithms for high performance, wide-area distributed file downloads. Parallel Process. Lett. **13**(2) (2003)
40. Plank, J., Beck, M., Elwasif, W., Moore, T., Swany, M., Wolski, R.: The internet backplane protocol: storage in the network. In: Proceedings of the Network Storage Symposium (1999)
41. Plank, J., Buchsbaum, A., Collins, R., Thomason, M.: Small parity-check erasure codes - exploration and observations. In: Proceedings of the International Conference on Dependable Systems and Networks (2005)
42. Plank, J., Thomason, M.: A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In: Proceedings of the International Conference on Dependable Systems and Networks (2004)
43. Rizzo, L.: Effective erasure codes for reliable computer communication protocols. SIGCOMM Comput. Commun. Rev. **27**(2) (1997)
44. Sarkar, P., Hartman, J.: Efficient cooperative caching using hints. In: Proceeding of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI) (1996)
45. Schmuck, F., Haskin, R.: GPFS: a shared-disk file system for large computing clusters. In: Proceedings of the First Conference on File and Storage Technologies (2002)
46. Schulzrinne, H., Rao, A., Lanphier, R.: Real time streaming protocol (rtsp). http://www.ietf.org/rfc/rfc2326.txt (1998)
47. Sloan digital sky survey: http://www.sdss.org (2005)
48. Seamons, K.E., Chen, Y., Jones, P., Jozwiak, J., Winslett, M.: Server-directed collective I/O in Panda. In: Proceedings of Supercomputing '95 (1995)
49. Sen, S., Rexford, J., Towsley, D.: Proxy prefix caching for multimedia streams. In: Proceedings of the IEEE INFOCOM Conference (1999)
50. Seti@home: The search for extraterrestrial intelligence. http://setiathome.ssl.berkeley.edu/ (2009)
51. Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup service for internet applications. In: Proceedings of the ACM SIGCOMM Conference (2001)
52. Strickland, J., Freeh, V., Ma, X., Vazhkudai, S.: Governor: autonomic throttling for aggressive idle resource scavenging. In: Proceedings of the 2nd IEEE International Conference on Autonomic Computing (2005)
53. Szalay, A.: Data explosion: astrophysics with terabytes of data. In: Proceedings of the 1st Scientific Data Intensive Computing Workshop (2004)
54. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems (1999)
55. Tierney, B., Gunter, D., Lee, Stoufer, M.: Enabling network-aware applications. In: Proceedings of the IEEE High Performance Distributed Computing conference (2001)
56. Tierney, B., Lee, J., Holding, M., Hylton, J., Drake, F.: A network-aware distributed storage cache for data intensive environments. In: Proceedings of the IEEE High Performance Distributed Computing conference (HPDC-8) (1999)
57. Vazhkudai, S., Ma, X., Freeh, V., Strickland, J., Tammineedi, N., Scott, S.: Freeloader: scavenging desktop storage resources for bulk, transient data. In: Proceedings of Supercomputing (2005)
58. Vazhkudai, S., Ma, X., Freeh, V., Strickland, J., Tammineedi, N., Simon, T.A., Scott, S.L.: Constructing collaborative desktop storage caches for large scientific datasets. ACM Transactions on Storage (TOS) **2**(3), 221–254 (2006)
59. Weatherspoon, H., Kubiatowicz, J.: Erasure coding vs. replication: a quantitative comparison. In: Proceedings of the 1st International Workshop on Peer-to-Peer Systems (2002)
60. Young, N.: On-line file caching. In: Proceedings the ACM-SIAM Symposium on Discrete Algorithms (SODA) (1998)