Transparent Fault Tolerance for Job Input Data in HPC Environments

Chao Wang[1], Sudharshan S. Vazhkudai[1], Xiaosong Ma[2,3], and Frank Mueller[3]

[1] National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN
[2] Qatar Computing Research Institute, Doha, Qatar
[3] Department of Computer Science, North Carolina State University Raleigh, NC
{wangcn, vazhkudaiss}@ornl.gov, {ma, mueller}@cs.ncsu.edu

January 14, 2014

Abstract

As the number of nodes in high-performance computing environments keeps increasing, faults are becoming common place causing losses in intermediate results of HPC jobs. Furthermore, storage systems providing job input data have been shown to consistently rank as the primary source of system failures leading to data unavailability and job resubmissions.

This chapter presents transparent techniques to improve the reliability, availability and performance of HPC I/O systems, for the job input data. In this area, the chapter contributes (1) a mechanism for offline job input data reconstruction to ensure availability of job input data and to improve center-wide performance at no cost to job owners; (2) an approach to automatic recover job input data at run-time during failures by recovering staged data from an original source; and (3) "just in time" replication of job input data so as to maximize the use of supercomputer cycles.

Experimental results demonstrate the value of these advanced fault tolerance techniques to increase fault resilience in HPC environments.

1 Introduction

Recent progress in high-performance computing (HPC) has resulted in remarkable Terascale systems with 10,000s or even 100,000s of processors. At such large counts of compute nodes, faults are becoming common place. Reliability data of contemporary systems illustrates that the mean time between failures (MTBF) / interrupts (MTBI) is in the range of 6.5-40 hours depending on the maturity / age of the installation [22]. Table 1 presents an excerpt from a recent study by Department of Energy (DOE) researchers that summarizes the reliability of several state-of-the-art supercomputers and distributed computing systems [21, 24]. The most common causes of

Table 1: Reliability of HPC Clusters

System	# Cores	MTBF/I	Outage source
ASCI Q	8,192	6.5 hrs	Storage, CPU
ASCI White	8,192	40 hrs	Storage, CPU
PSC Lemieux	3,016	6.5 hrs	
Google	15,000	20 reboots/day	Storage, memory
Jaguar@ORNL	23,416	37.5 hrs	Storage, memory

failure are processor, memory and storage errors / failures. When extrapolating for current systems in such a context, the MTBF for peta-scale systems is predicted to be as short as 1.25 hours [31].

Furthermore, data and input/output (I/O) availability are integral to providing non-stop, continuous computing capabilities to MPI (Message Passing Interface) applications. Table 1 indicates that the storage subsystem is consistently one of the primary sources of failure on large supercomputers. This situation is only likely to worsen in the near future due to the growing relative size of the storage system forced by two trends: (1) disk performance increases slower than that of CPUs and (2) users' data needs grow faster than does the available

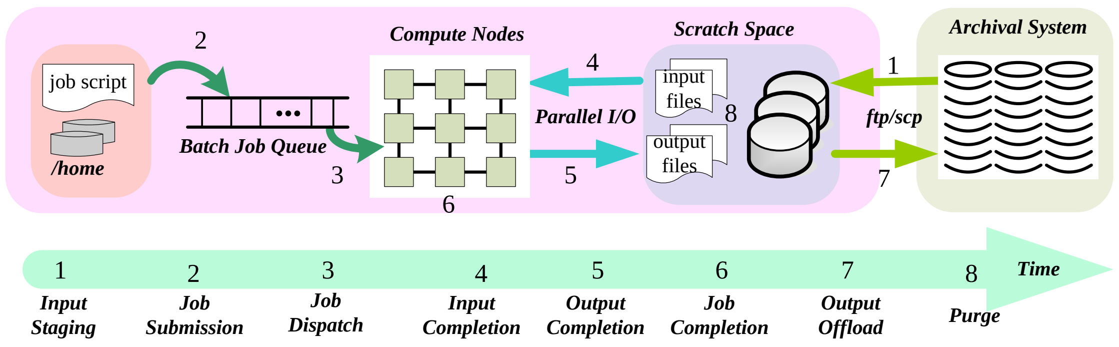


Figure 1: Event timeline of a typical MPI job's data life-cycle

compute power [19]. A survey of DOE applications suggests that most applications require a sustained 1 GB/sec I/O throughput for every TeraFlop of peak computing performance. Thus, a PetaFlop computer will require 1 TB/sec of I/O bandwidth. The Roadrunner system at the Los Alamos National Laboratory, which was the world's fastest supercomputer on the top500 list in June of 2008 and the first to break the PetaFlop barrier, has 216 I/O nodes attaching 2048 TB of storage space [1]. An I/O subsystem of this size makes a large parallel machine itself and is subject to very frequent failures.

To prevent valuable computation to be lost due to failures, fault tolerance (FT) techniques must be provided to ensure that HPC jobs can make progress in the presence of failures.

1.1 Background

This section introduces the storage hierarchies and I/O in most modern HPC systems.

Figure 1 gives an overview of an event timeline describing a typical supercomputing job's data life-cycle. Users stage their job input data from elsewhere to the scratch space, submit their jobs using a batch script, and offload the output files to archival systems or local clusters.

Here, we can divide the storage system into four levels:

1. **Primary storage** is the memory of the compute nodes. Main memory is directly or indirectly connected to the CPU of compute node via a memory bus. The I/O device at this level is actually comprised of two memory buses: an address bus and a data bus.

2. **Secondary storage** is the local disk of compute node. It is not directly accessible by the CPU. I/O channels are used by the compute node to access secondary storage. However, some HPC systems, such as BG/L, have no local disk.

3. **The scratch space** of an HPC system is a storage subsystem. It is managed by a parallel file system, such as Lustre, and is connected to the compute node (clients of the PFS) through I/O nodes and network connections.

4. **The archival system**, also known as disconnected storage, is used to backup data. The archival system increases general information security, since it is unaffected by the failure of compute nodes and cannot be affected by computer-based security attacks. Also, archival systems are less expensive than scratch space storage. The I/O between the compute systems and the archival systems is typically through network transfer protocols, such as ftp and scp, intervened by human beings or system schedulers through job scripts.

HPC systems consume and produce massive amounts of data in a short time, which turns the compute-bound problems into an I/O-bound problem. However, HPC storage systems are becoming severe bottlenecks in current extreme-scale supercomputers in both reliability and performance. First of all, storage systems consistently rank as the primary source of system failures, as depicted in Table 1. Second, I/O performance is lagging behind. Much work on the multi-level storage and I/O in HPC environments is needed to ensure that the data can be transferred, stored and retrieved quickly and correctly.

1.2 Motivation

In today's large-scale HPC environment, MPI jobs generally need to read input data. Jobs are interrupted or rerun if job input data is unavailable or lost. However, storage systems providing job input data have been shown to consistently rank as the primary source of system failures, according to logs from large-scale parallel computers and commercial data centers [21]. This trend is only expected to continue as individual disk bandwidth grows much slower than the overall supercomputer capacity. Therefore, the number of disk drives used in a supercomputer will need to increase faster than the overall system size. It is predicted that by 2018, a system at the top of the top500.org chart will have more than 800,000 disk drives with around 25,000 disk failures per year [35].

Thus, coping with failures for job input data is a key issue as we scale to Peta- and Exa-flop supercomputers. In this chapter, for the job input data, we provide offline recovery, online recovery, and temporal replication to improve the reliability, availability and performance of HPC I/O systems.

1.3 Contributions

The objective of this work is to improve the reliability, availability and performance of the job input data through the following contributions:

1. We explore a mechanism of offline job input data reconstruction to ensure availability of job input data and to improve center-wide performance. With data source information transparently extracted from the staging request, our framework allows jobs to be scheduled even when parts of their prestaged input data are unavailable due to storage system failures. This is accomplished through transparent data reconstruction at no cost to job owners.

2. We present an approach to automatic recover job input data during failures by recovering staged data from an original source. The proposed mechanism captures I/O failures (e.g., due to storage node failures) while a running job is accessing its input file(s) and retrieves portions of the input file that are stored on the failed nodes. We further deploy this technique in current high performance computing systems to recover the input data of applications at run-time. The implementation of the mechanism is realized within the Lustre parallel file system. An evaluation of the system using real machines is also presented.

3. We propose "just in time" replication of job input data so as to maximize the use of supercomputer cycles. The mechanism adds selective redundancy to job input data by synergistically combining the parallel file system with the batch job scheduler to perform temporal replication transparently. These replicas will be used for fast data recovery during a job execution and are subsequently removed when the job completes or finishes consuming the input data.

   We show that the overall space and I/O bandwidth overhead of temporal replication is a reasonable small fraction of the total scratch space on modern machines, which can be further improved by optimizations to shorten the replication duration.

1.3.1 Offline Reconstruction of Job Input Data

Our target environment is that of shared, large, supercomputing centers. In this setting, vast amounts of *transient* job data routinely passes through the scratch space, hosted on parallel file systems. Supercomputing jobs, mostly parallel time-step numerical simulations, typically initialize their computation with a significant amount of staged input data.

We propose *offline data reconstruction* that transparently verifies the availability of the staged job input data and fetches unavailable pieces from external data sources in case of storage failures. Our approach takes advantage of the existence of an external data source/sink and the immutable nature of job input data. Collectively, from a center standpoint, these techniques globally optimize resource usage and increase data and service availability. From a user job standpoint, they reduce job turnaround time and optimize the usage of allocated time.

Periodic data availability checks and transparent data reconstruction is performed to protect the staged data against storage system failures. Compared to existing approaches, this technique incurs minimum user operational cost, low storage usage cost, and no computation time cost. It also reduces the average job waiting time of the entire system and increases overall center throughput and service.

We implement offline recovery schemes into parallel file systems so that applications can seamlessly utilize available storage elements.

1.3.2 On-the-fly Recovery of Job Input Data

Storage system failure is a serious concern as we approach Petascale computing. Even at today's sub-Petascale levels, I/O failure is the leading cause of downtimes and job failures.

We contribute a novel, on-the-fly recovery framework for job input data into supercomputer parallel file systems. The framework exploits key traits of the HPC I/O workload to reconstruct lost input data during job execution from remote, immutable copies. Each reconstructed data stripe is made immediately accessible in the client request order due to the delayed metadata update and fine-granular locking while unrelated accesses to the same file remain unaffected.

We implement the recovery component within the Lustre parallel file system, thus building a novel application-transparent online recovery solution. Our solution is integrated into Lustre's two-level locking scheme using a two-phase blocking protocol.

1.3.3 Temporal Replication of Job Input Data

Storage systems in supercomputers are a major reason for service interruptions. RAID (Redundant Arrays of Inexpensive Disks) solutions alone cannot provide sufficient protection as 1) growing average disk recovery times make RAID groups increasingly vulnerable to disk failures during reconstruction, and 2) RAID does not help with higher-level faults such as failed I/O nodes.

We present a complementary approach based on the observation that files in the supercomputer scratch space are typically accessed by batch jobs whose execution can be anticipated. Therefore, we propose to transparently, selectively, and temporarily replicate "active" job input data by coordinating parallel file system activities with those of the batch job scheduler. We implement a temporal replication scheme in the popular Lustre parallel file system and evaluate it with real-cluster experiments. Our results show that the scheme allows for fast online data reconstruction with a reasonably low overall space and I/O bandwidth overhead.

2 Offline Reconstruction of Job Input Data

2.1 Introduction

Recovery operations are necessary to reconstruct pre-staged data, in the event of storage system failure before the job is scheduled. These operations are performed as periodic checks to ensure data availability for any given job. Further, the recovery is performed as part of system management even before the job is scheduled. Therefore, it is not charged against the users' compute time allocation. In the absence of reconstruction, user jobs are left to the mercy of traditional system recovery or put back in the queue for rescheduling.

To start with, this mode of recovery is made feasible due to the transient nature of job data and the fact that they have immutable, persistent copies elsewhere. Next, many high-performance data transfer tools and protocols (such as hsi [18] and GridFTP [7]) support partial data retrieval, through which disjoint segments of missing file data—specified by pairs of start offset and extent—can be retrieved. Finally, the costs of network transfer is decreasing much faster than that of disk-to-disk copy. These reasons, collectively, enable and favor offline recovery.

A staged job input data on the scratch parallel file system may have originated from the "/home" area or an HPSS archive [15] in the supercomputer center itself or from an end-user site. To achieve proactive data reconstruction, however, we need rich metadata about data source and transfer protocols used for staging. In addition, we need sophisticated recovery mechanisms built into parallel file systems.

2.2 Architecture

2.2.1 Metadata: Recovery Hints

To enable offline data recovery, we first propose to extend the parallel file system's metadata with *recovery information*, which can be intelligently used to improve fault tolerance and data/resource availability. Staged input data has persistent origins. Source data locations, as well as information regarding the corresponding data movement protocols, can be recorded as optional recovery metadata (using the extended attributes feature) on file systems. For instance, the location can be specified as a uniform resource index (URI) of the dataset, comprised of the protocol, URL, port and path (e.g., "http://source1/StagedInput" or "gsiftp://mirror/StagedInput"). In addition to URIs, user credentials, such as GSI (Grid Security Infrastructure) certificates, needed to access the particular dataset from remote mirrors can also be included as metadata so that data recovery can be initiated on behalf of the user. Simple file system interface extensions (such as those using extended attributes) would allow the capture of this metadata. We have built mechanisms for the recovery metadata to be automatically stripped from a job submission script's staging/offloading commands, facilitating transparent data recovery. One advantage of embedding such recovery-related information in file system metadata is that the description of a user job's data "source" and "sink" becomes an integral part of the transient dataset on the supercomputer while it executes. This allows the file system to recover elegantly without manual intervention from the end users or significant code modification.

2.2.2 Data Reconstruction Architecture

In this section, we present a prototype recovery manager. The prototype is based on the Lustre parallel file system [13], which is being adopted by several leadership-class supercomputers. A Lustre file system comprises of the following three key components: *Client*, *MDS* (MetaData Server) and *OSS* (Object Storage Server). Each OSS can be configured to host several *OST*s (Object Storage Target) that manage the storage devices (e.g., RAID storage arrays).

The reconstruction process is as follows. 1) Recovery hints about the staged data are extracted from the job script. Relevant recovery information for a dataset is required to be saved as recovery metadata in the Lustre file system. 2) The availability of staged data is periodically checked (i.e., check for OST failure). This is performed after staging and before job scheduling. 3) Data is reconstructed after OST failures. This involves finding spare OSTs to replace the failed ones, orchestrating the parallel reconstruction, fetching the lost data stripes from the data source, using recovery hints and ensuring the metadata map is up-to-date for future accesses to the dataset.

Extended Metadata for Recovery Hints: As with most parallel file systems, Lustre maintains file metadata on the MDS as inodes. To store recovery metadata for a file, we have added a field in the file inode named "recov", the value of which is a set of URIs indicating permanent copies of the file. The URI is a character string, specified by users during data staging. Maintaining this field requires little additional storage (less than 64 bytes) and minimal communication costs for each file. We have also developed two additional Lustre commands, namely *lfs setrecov* and *lfs getrecov* to set and retrieve the value of the "recov" field respectively, by making use of existing Lustre system calls.

I/O Node Failure Detection: To detect the failure of an OST, we use the corresponding Lustre feature with some extensions. Instead of checking the OSTs sequentially, we issue commands to check each OST in parallel. If no OST has failed, the probe returns quickly. Otherwise, we wait for a Lustre-prescribed timeout of 25 seconds. This way, we identify all failed OSTs at once. This check is performed from the head node, where the scheduler also runs.

File Reconstruction: The first step towards reconstruction is to update the stripe information of the target file, which is maintained as part of the metadata. Each Lustre file f is striped in a round-robin fashion over a set of m_f OSTs, with indices $T_f = \{t_0, t_1, \ldots, t_{m_f}\}$. Let us suppose OST f_i is found to have failed. We need to find another OST as a substitute from the stripe list, T_f. To sustain the performance of a striped file access, it is important to have a distinct set of OSTs for each file. Therefore, in our reconstruction, an OST that is not originally in T_f will take OST f_i's position, whenever at least one such OST is available. More specifically, a list L of indices of all available OSTs is compared with T_f and an index $t_{i'}$ is picked from the set $L - T_f$. We are able to achieve this, since Lustre has a global index for each OST. The new stripe list for the dataset is $T'_f = \{t_0, t_1, \ldots, t_{i-1}, t_{i'}, t_{i+1}, \ldots, t_{m_f}\}$. When a client opens a Lustre file for the first time, it will obtain the stripe list from the MDS and create a local copy of the list, which it will use for subsequent read/write calls. In our reconstruction scheme, the client that locates the failed OST updates its local copy of the stripe and sends a message to the MDS to trigger an update of the master copy. The MDS associates a dirty bit with the file, indicating the availability of updated data to other clients. A more efficient method would be to let the MDS multicast a message to all clients that have opened the file, instructing them to update their local copies of the stripe list. This is left as future work.

When the new stripe list T'_f is generated and distributed, storage space needs to be allocated on OST $t_{i'}$ for the data previously stored on OST t_i. As an object-based file system [28], Lustre uses objects as storage containers on OSTs. When a file is created, the MDS selects a set of OSTs that the file will be striped on and creates an object on each one of them to store a part of the file. Each OST sends back the id of the object that it creates and the MDS collects the object ids as part of the stripe metadata. The recovery manager works in the same way, except that the MDS only creates an object in OST $t_{i'}$. The id of the object on the failed OST t_i is replaced with the id of the object created on OST $t_{i'}$. The length of an object is variable [28] and, therefore, the amount of data to be patched is not required to be specified at the time of object creation.

In order to patch data from a persistent copy and reconstruct the file, the recovery manager needs to know which specific byte ranges are missing. It obtains this information by generating an array of $\{offset, size\}$ pairs according to the total number of OSTs used by this file, m_f, the position of the failed OST in the stripe list, i, and the stripe size, $ssize$. Stripe size refers to the number of data blocks. Each $\{offset, size\}$ pair specifies a chunk of data that is missing from the file. Given the round-robin striping pattern used by Lustre, it can be calculated that for a file with size $fsize$, each of the first $k = fsize \bmod ssize$ OSTs will have $\lceil \frac{fsize}{ssize} \rceil$ stripes and each of the other OSTs will have $\lfloor \frac{fsize}{ssize} \rfloor$ stripes. For each OST, it can be seen that in the j_{th} pair, $offset = j \times ssize$, and in each pair except the last one, $size = ssize$. If the OST has the last stripe of the file, then $size$ will be smaller in the last pair.

The recovery manager then acquires the URIs to remote permanent copies of the file from the "recov" field of the file inode, as we described above. Then, it establishes a connection to the remote data source, using the protocol specified in the URI, to patch each chunk of data in the array of $\{offset, size\}$ pairs. These are then written to the object on OST $t_{i'}$. We have built mechanisms so that the file reconstruction process can be conducted from the head node or from the individual OSS nodes (to exploit parallelism), depending on transfer tool and Lustre client availability.

Patching Session: At the beginning of each patching operation, a session is established between the patching nodes (the head node or the individual OSS nodes) and the data source. Many data sources assume downloads occur in an interactive session that includes authentication, such as GridFTP [7] servers using UberFTP client [2] and HPSS [15] using hsi [18]. In our implementation, we use Expect [25], a tool specifically geared towards automating interactive applications, to establish and manage these interactive sessions. We used Expect so that authentications and subsequent partial retrieval requests to the data 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.

2.3 Experimental Setup

In our performance evaluation, we assess the effect of storage node failures and its subsequent data reconstruction overhead on a real cluster using several local or remote data repositories as sources of data staging.

Our testbed for evaluating the proposed data reconstruction approach is a 40-node cluster at Oak Ridge National Laboratory (ORNL). Each machine is equipped with a single 2.0GHz Intel Pentium 4 CPU, 768 MB of main memory, with a 10/100 Mb Ethernet interconnection. The operating system is Fedora Core 4 Linux with a Lustre-patched kernel (version 2.6.12.6) and the Lustre version is 1.4.7.

Since our experiments focus on testing the server-side of Lustre, we have setup the majority of the the cluster nodes as I/O servers. More specifically, we assign 32 nodes to be OSSs, one node to be the MDS and one node to be the client. The client also doubles as the head node. In practice, such a group of Lustre servers will be able to support a fairly large cluster of 512-2048 compute nodes (with 16:1-64:1 ratios seen in contemporary supercomputers).

We used three different data sources to patch pieces of a staged input file: (1) An NFS server at ORNL that resides outside the subnet of our testbed cluster ("Local NFS") (2) an NFS server at North Carolina State University ("Remote NFS") and (3) a GridFTP [8] server ("GridFTP") with a PVFS [12] backend on the TeraGrid Linux cluster at ORNL, outside the Lab's firewall, accessed through the UberFTP client interface [2].

2.4 Performance of Transparent Input Data Reconstruction

As illustrated in Section 2.2, for each file in question, the reconstruction procedure can be divided into the following steps. 1) The failed OSTs are determined by querying the status of each OST. 2) The file stripe metadata is updated after replacing the failed OST with a new one. 3) The missing data is patched in from the source. This involves retrieving the URI information of the data source from the MDS, followed by fetching the missing data stripes from the source and subsequently patching the Lustre file to complete the reconstruction. These steps are executed sequentially and atomically as one transaction, with concurrent accesses, to the file in question, protected by file locking. The costs of the individual steps are independent of one another, *i.e.*, the cost of reconstruction is precisely the sum of the costs of the steps. Below we discuss the overhead of each step in more detail.

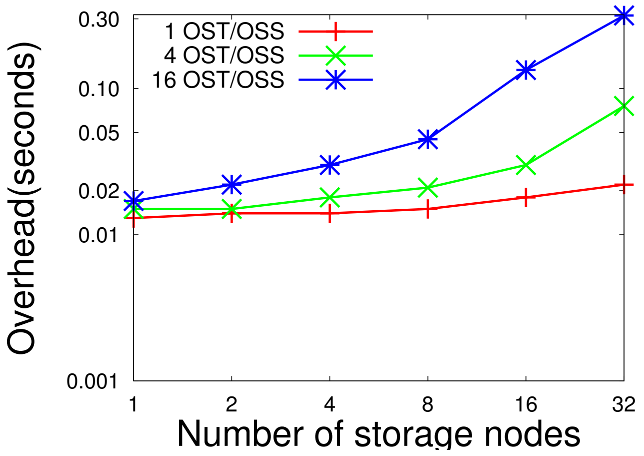


Figure 2: Cost of finding failed OSTs

In the first step, all the OSTs are checked in parallel. The cost of this check grows with the number of OSTs due to the fact that more threads are needed to check the OSTs. Also, the metadata file containing the status of all OSTs will be larger, which increases the parsing time. This step induces negligible network traffic by sending small status checking messages. The timeout to determine if an OST has failed is the default value of 25 seconds in our experiments. Shorter timeouts might result in false positives.

Figure 2 shows the results of benchmarking the cost of step 1 (detecting OST failures) when there are no such failures. In other words, this is the cost already observed in the majority of file accesses, where data unavailability is not an issue. In this group of tests, we varied the number of OSTs per I/O node (OSS), which in reality is configured by system administrators. High-end systems tend to have multiple OSTs per OSS, medium-sized or smaller clusters often choose to use local disks and have only one OST per OSS. From Figure 2, it can be seen that the overhead of step 1 is fairly small with a moderate number of OSTs. In most cases, the step 1 cost is under 0.05 seconds. However, this overhead grows sharply as the number of OSTs increases to 256 (16 OSTs on each of the 16 OSSs) and 512 (16 OSTs on each of the 32 OSSs). This can be attributed to the network congestion caused by the client communicating with a large number of OSTs in a short span. Note that the current upper limit of OSTs allowed by Lustre is 512, which incurs an overhead of 0.3 seconds, which is very small considering this is a one-time cost for input files only.

The second recovery step has constant cost regardless of the number of OSTs as the only parties involved in this operation are the *MDS* and the *client* that initiates the file availability check (in our case, the head node). In our experiments, we also assessed this overhead and found it to be in the order of milliseconds. Further, this remains constant regardless of the number of OSSs/OSTs. The cost due to multiple *MDS* (normally two for Lustre) is negligible.

The overhead of the third recovery step is expected to be the dominant factor in the reconstruction cost when OST failures do occur. Key factors contributing to this cost are the amount of data and the layout of missing data in the file. It is well known that non-contiguous access of data will result in lower performance than sequential accesses. The effect of data layout on missing data is more significant if sequential access devices such as tape

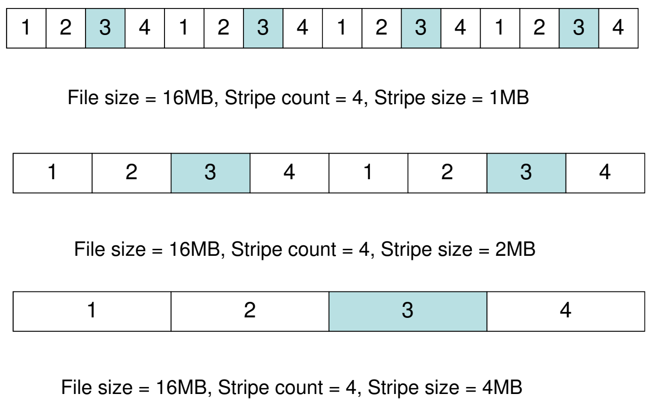
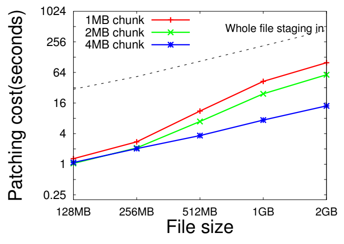
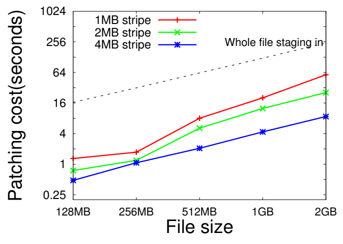

1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
File size = 16MB, Stripe count = 4, Stripe size = 1MB
1 2 3 4 1 2 3 4
File size = 16MB, Stripe count = 4, Stripe size = 2MB
1 2 3 4
File size = 16MB, Stripe count = 4, Stripe size = 4MB

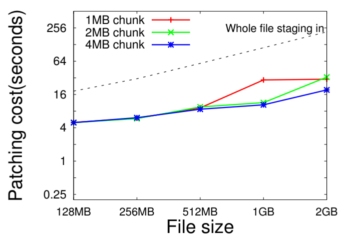
Figure 3: Round-robin striping over 4 OSTs



(a) Remote NFS



(b) Local NFS



(c) GridFTP

Figure 4: Patching costs for single OST failure from a stripe count 32. Filesize/32 is stored on each OST and that is the amount of data patched. Also shown is the cost of staging the entire file again, instead of just patching a single OST worth of data.

drives are used at the remote data source. More specifically, factors that may affect patching cost include the size of each chunk of missing data, the distance between missing chunks, and the offset of the first missing chunk. Such a layout is mostly determined by the striping policy used by the file. To give an example, Figure 3 illustrates the different layout of missing chunks when OST 3 fails. The layout results from different striping parameters, such as the *stripe width* (called "stripe count" in Lustre terminology) and the *stripe size*. The shaded boxes indicate missing data stripes (chunks).

We conducted multiple sets of experiments to test the data patching cost using a variety of file striping settings and data staging sources. Figures 4(a)-(c) show the first group of tests, one per data staging source (Remote NFS, Local NFS, and GridFTP). Here, we fixed the stripe count (stripe width) at 32 and increased the file size from 128MB to 2GB. Three popular stripe sizes were used with 1MB, 2MB, and 4MB chunks as the stripe unit, respectively. To inject a failure, one of the OSTs is manually disconnected so that 1/32 of the file data is missing. The *y-axis* measures the total patching time *in log scale*. For reference, a dotted line in each figure shows the time it takes to stage-in the whole file from the same data source (in the absence of our enhancements).

We notice that the patching costs from NFS servers are considerably higher with small stripe sizes (*e.g.*, 1MB). As illustrated in Figure 3, a smaller stripe size means the replacement OST will retrieve and write a larger number of non-contiguous data chunks, which results in higher I/O overhead. Interestingly, the patching costs from the GridFTP server remains nearly constant with different stripe sizes. The GridFTP server uses PVFS, which has better support than NFS for non-contiguous file accesses. Therefore, the time spent obtaining the non-contiguous chunks from the GridFTP server is less sensitive to smaller stripe sizes. However, both reconstructing from NFS and GridFTP utilize standard POSIX/Linux I/O system calls to seek in the file at the remote data server. In addition, there is also seek time to place the chunks on the OST. A more efficient method would be to directly rebuild a set of stripes on the native file system of an OST, which avoids the seek overhead that is combined with redundant read-ahead caching (of data between missing stripes) when rebuilding a file at the Lustre level. This, left as future work, should result in nearly uniform overhead approximating the lowest curve regardless of stripe size.

Overall, we have found that the file patching cost scales well with increasing file sizes. For the NFS servers, the patching cost using 4MB stripes is about 1/32 of the entire-file staging cost. For the GridFTP server, however, the cost appears to be higher and less sensitive to the stripe size range we considered. The reason is that GridFTP is tuned for bulk data transfers (GB range) using I/O on large chunks, preferably tens or hundreds of MBs [27]. Also, the GridFTP performance does improve more significantly, compared with other data sources, as the file size increases.

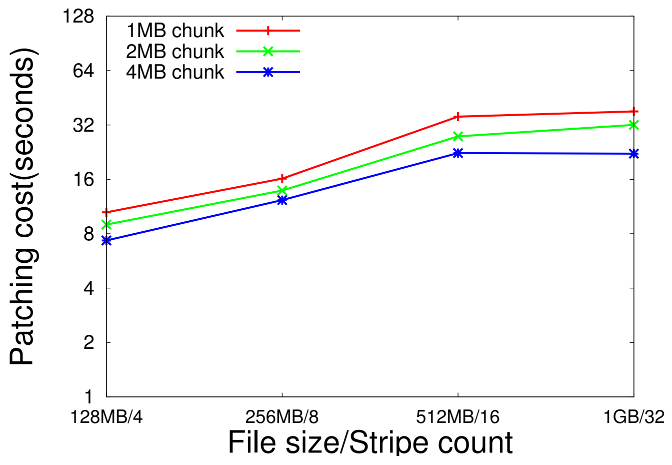
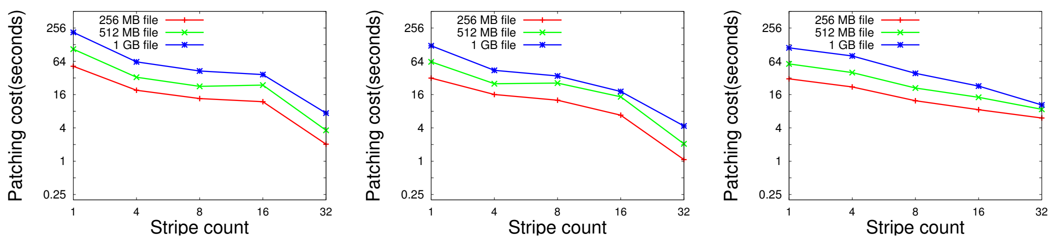


Figure 5: Patching from local NFS. Stripe count increases with file size. One OST fails and its data is patched.

Figures 5 and 6 demonstrate the patching costs with different stripe counts. In Figure, 5 we increase the stripe count at the same rate as the file size to show how the patching cost from the local NFS server varies. The amount of missing data due to an OST failure is $\frac{1}{stripe\_count} \times file\_size$. Therefore, we patch the same amount of data for each point in the figure. It can be seen that the patching cost grows as the file size reaches 512MB and remains constant thereafter. This is caused by the fact that missing chunks in a file are closer to each other with a smaller stripe count. Therefore, when one chunk is accessed from the NFS server, it is more likely for the following chunks to be read ahead into the server cache. With larger file sizes (512MB or more), distances between the missing chunks are larger as well. Hence, the server's read-ahead has no effect. Figure 6 shows the results of fixing the stripe size at 4MB and using different stripe counts. Note that a stripe count of 1 means transferring the entire file. From this figure, we can see how the cost of reconstructing a certain file decreases as the file is striped over more OSTs.

2.5 Conclusion

We have proposed a novel way to reconstruct transient, job input data in the face of storage system failure. From a user's standpoint, our techniques help reduce the job turnaround time and enable the efficient use of the precious, allocated compute time. From a center's standpoint, our techniques improve serviceability by reducing the rate of resubmissions due to storage system failure and data unavailability. By automatically reconstructing missing input data, a parallel file system will greatly reduce the impact of storage system failures on the productivity of both supercomputers and their users.



(a) Patching from remote NFS server (b) Patching from local NFS server (c) Patching from GridFTP server

Figure 6: Patching costs with stripe size 4MB

3 On-the-fly Recovery of Job Input Data

3.1 Introduction

In HPC settings, data and I/O availability is critical to center operations and user serviceability. Petascale machines require 10,000s of disks attached to 1,000s of I/O nodes. Plans for 100k to 1M disks are being discussed in this context. The numbers alone imply severe problems with reliability. In such a setting, failure is inevitable. I/O failure and data unavailability can have significant ramifications to a supercomputer center at large. For instance, an I/O node failure in a parallel file system (PFS) renders portions of the data inaccessible resulting in either application stalling on I/O or being forced to be resubmitted and rescheduled.

Upon an I/O error, the default behavior of file systems is to simply propagate the error back to the client. Usually, file systems do little beyond providing diagnostics so that the application or the user may perform error handling and recovery. For applications that go through rigid resource allocation and lengthy queuing to execute on Petascale supercomputers, modern parallel file systems' failure to mask storage faults appears particularly expensive.

Standard hardware redundancy techniques, such as RAID, only protect against entire disk failures. Latent sector faults (occurring in 8.5% of a million disks studied [5]), controller failures, or I/O node failures can render data inaccessible even with RAID. Failover strategies require spare nodes to substitute the failed ones, an expensive option with thousands of nodes. It would be beneficial to address these issues *within the file system* to provide graceful, transparent, and portable data recovery.

HPC environments provide unique fault-tolerance opportunities. Consider a typical HPC workload. Before submitting a job, users stage in data to the scratch PFS from end-user locations. After the job dispatch (hours to days later) and completion (again hours or days later), users move their output data off the scratch PFS (*e.g.*, to their local storage). Thus, job input and output data seldom need to reside on the scratch PFS beyond a short window before or after the job's execution. Specifically, key characteristics of job *input* data are their being (1) transient, (2) immutable, and (3) redundant in terms of a remote source copy.

We propose *on-the-fly data reconstruction* during job execution. We contribute an application-transparent extension to the widely used Lustre parallel file system [13], thereby adding reliability into the PFS by shielding faults at many levels of an HPC storage system from the applications. With our mechanism, a runtime I/O error (EIO) captured by the PFS instantly triggers the recovery of missing pieces of data and resolves application requests immediately when such data becomes available.

Such an approach is a dramatic improvement in fault handling in modern PFSs. At present, an I/O error is propagated through the PFS to the application, which has no alternative but to exit. Users then need to re-stage input files if necessary and resubmit the job. Instead of resource-consuming I/O node failover or data replication to avoid such failures, our solution does not require additional storage capacity. Only the missing data stripes residing on the failed I/O node are staged again from their original remote location. Exploiting Lustre's two-level locks, we have implemented a two-phase blocking protocol combined with delayed metadata updates that allows unrelated data requests to proceed while outstanding I/O requests to reconstructed data are served in order, as soon as a stripe becomes available. Recovery can thus be overlapped with computation and communication as stripes are recovered. Our experimental results reinforce this by showing that the increase in job execution time due to on-the-fly recovery is negligible compared to non-faulting runs.

Consider the ramifications of our approach. From a center standpoint, I/O failures traditionally increase the overall *expansion factor*, *i.e.*, *(wall_time + wait_time)/wall_time* averaged over all jobs (the closer to 1, the better). Many federal agencies (DOD, NSF, DOE) are already requesting such metrics from HPC centers. From a user standpoint, I/O errors result in dramatically increased turnaround time and, depending on already performed computation, a corresponding waste of resources. Our method significantly reduces this waste and results in lower expansion factors.

3.2 On-the-fly Recovery

The overarching goal of this work is to address file systems' fault tolerance when it comes to serving HPC workloads. The following factors weigh in on our approach.

(1) Mitigate the effects of I/O node failure: An I/O node failure can adversely affect a running job by causing it to fail, being requeued or exceeding time allocation, all of which impacts the HPC center and user. Our solution promotes continuous job execution that minimizes the above costs. *(2) Improve file system response to failure:* File system response to failure is inadequate. As we scale to thousands of I/O nodes and few orders of magnitude more disks, file systems need to be able to handle failure gracefully. *(3) Target HPC workloads:* The transient and immutable nature of job input data and its persistence at a remote location present an unique opportunity to address data availability in HPC environments. We propose to integrate fault tolerance into the PFS specifically for HPC I/O workloads. *(4) Be inclusive of disparate data sources and protocols:* HPC users use a variety of storage systems and transfer protocols to host and move their data. It is desirable to consider external storage resources and protocols as part of a broader I/O hierarchy. *(5) Be transparent to client applications:* Applications are currently forced to explicitly handle I/O errors or to simply ignore them. We promote a recovery scheme widely transparent to the application. *(6) Performance:* For individual jobs, on-the-fly recovery should impose minimal overhead on existing PFS functionality. For a supercomputing center, it should improve the overall job throughput compared to requeuing the job.

3.2.1 Architectural Design

To provide fault tolerance to PFS, the on-the-fly recovery component should be able to successfully trap I/O error of a system call resulting from I/O node failure. In a typical parallel computing environment, parallel jobs are launched on the numerous compute nodes (tens of thousands), and each one of those processes on the compute nodes perform I/O. Figure 7 depicts the overall design. Each compute node can act as a client to the parallel file system. Upon capturing an I/O error from any of these compute nodes, data recovery is set in motion. The calling process is blocked, and so is any other client trying to access the same unavailable data. The recovery process consults the MDS of the PFS to obtain remote locations where persistent copies of the job input data reside. (We discuss below how this metadata is captured.) It then creates the necessary objects to hold the data stripes that are to be recovered. Using the recovery metadata, remote patching is performed to fetch the missing stripes from the source location. The source location could be "/home", or an HPSS archive in the same HPC center, or a remote server. The patched data is stored in the PFS, and the corresponding metadata for the dataset in question is updated in the MDS. More specifically, missing stripes are patched in the client request order. Subsequently, blocked processes resume their execution as data stripes become available. Thus, the patching of missing stripes not yet accessed by the client is efficiently overlapped with client I/O operations to significantly reduce overhead.

3.2.2 Automatic Capture of Recovery Metadata

To enable on-demand data recovery, we extend the PFS's metadata with recovery information. Staged input data has persistent origins. Source data locations, as well as information regarding the corresponding data movement protocols, are recorded as optional recovery metadata (using the extended attributes feature) on file systems. Locations are specified as a URI of the dataset comprised of the protocol, URL, port and path (e.g., http://source1/StagedInput or gsiftp://mirror/StagedInput). Simple file system interface extensions (*e.g.*, extended attributes) capture this metadata. We have built mechanisms for the recovery metadata to be automatically stripped from a job submission script's staging commands for *offline* recovery [44] that we utilize here for *online* recovery. By embedding such recovery-related information in file system metadata, the description of a user job's data source and sink becomes an integral part of the transient dataset on the supercomputer while it executes.

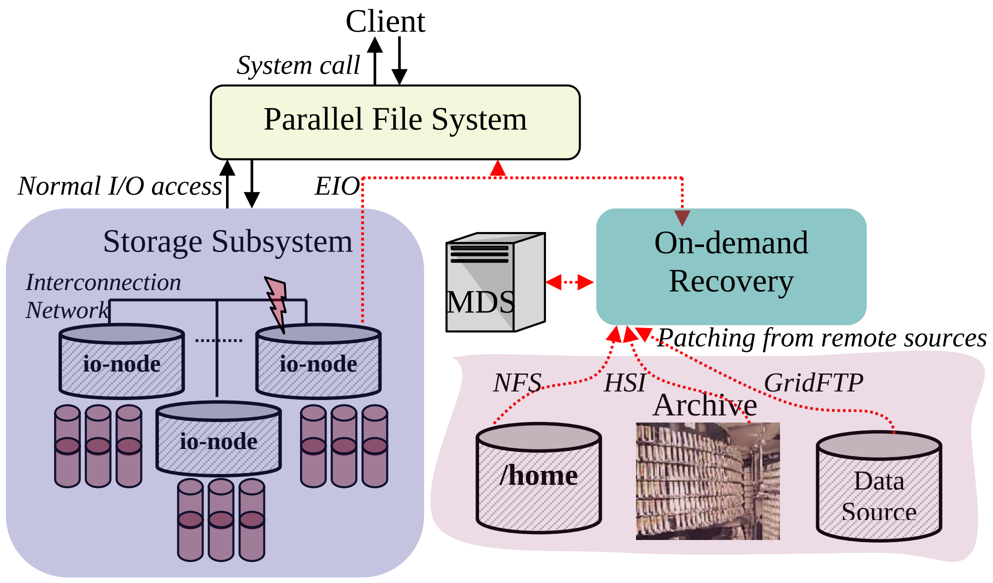


Figure 7: Architecture of on-the-fly recovery

User credentials, such as GSI certificates, may be needed to access the particular dataset from remote mirrors. These credentials can also be included as file metadata so that data recovery can be initiated on behalf of the user.

3.2.3 Impact on Center and User

Performance of online recovery requires further analysis. PFS at contemporary HPC centers can support several Gbps of I/O rate. However, this requires availability of all data and absence of failures in the storage subsystem. When faced with a RAID recoverable failure (e.g., an entire disk failure), file systems perform in either "degraded" or "rebuild" mode, both of which incur perceivable performance losses [38]. In cases where standard hardware-based recovery is not feasible, the only option is to trigger an application failure.

As application execution progresses, the performance impact (and potential waste of resources) due to failures increases resulting also in substantially increased turnaround time when a job needs to be requeued. These aspects also impact overall HPC center serviceability.

On-the-fly recovery offers a viable alternative in such cases. With ever increasing network speeds, HPC centers' connectivity to high-speed links, highly tuned bulk transport protocols are extremely competitive. For instance, ORNL's Leadership Class Facility (LCF) is connected to several national testbeds like TeraGrid (a 10Gbps link), UltrascienceNet, Lambda Rail, etc. Recent tests have shown that a wide-area Lustre file system over the TeraGrid from ORNL to Indiana University can offer data transfer speeds of up to 4.8 Gbps [36] for read operations bringing remote recovery well within reach.

Depending on how I/O is interspersed in the application, remote recovery has different merits. The majority of HPC scientific applications conduct I/O in a burst fashion by performing I/O and computation in distinct phases. These factors can be exploited to overlap remote recovery with computation and regular I/O requests. Once a failure is recognized and recovery initiated, the recovery process can patch other missing stripes of data that will eventually be requested by the application and not just the ones already requested. Such behavior can improve recovery performance significantly.

At other times, however, we may not be able to overlap recovery efficiently. In such cases, instead of consuming compute time allocation, a job might decide that being requeued is beneficial, thereby compromising on turnaround time. Thus, a combination of factors, such as I/O stride, time already spent on computation, cost of remote recovery and a turnaround time deadline, can be used to decide if and when to conduct remote data reconstruction. Nonetheless, the cause of I/O errors needs to be rectified before the next job execution. Our experiments analyze results and discuss their affect on job turnaround time in light of on-the-fly recovery.

3.3 Implementation

In this section, we illustrate how on-the-fly recovery has been implemented in Lustre FS. Should a storage failure occur due to an OSS or OST failure, the original input data can be replenished from the remote data source by reconstructing unavailable portions of files.

In supercomputers, remote I/O is usually conducted through the head or service nodes and, therefore, these nodes are likely candidates for the initiation of recovery. In our implementation, the head node of a supercomputer doubles as a recovery node and has a Lustre client installed on it. It schedules recovery in response to the requests received from the compute nodes, which observe storage failures upon file accesses. The head node serves as a coordinator that facilitates recovery management and streamlines reconstruction requests in a consistent and non-redundant fashion. Figure 8 depicts the recovery scenario. Events annotated by numbers happen consecutively in the indicated order resulting in four distinct phases.

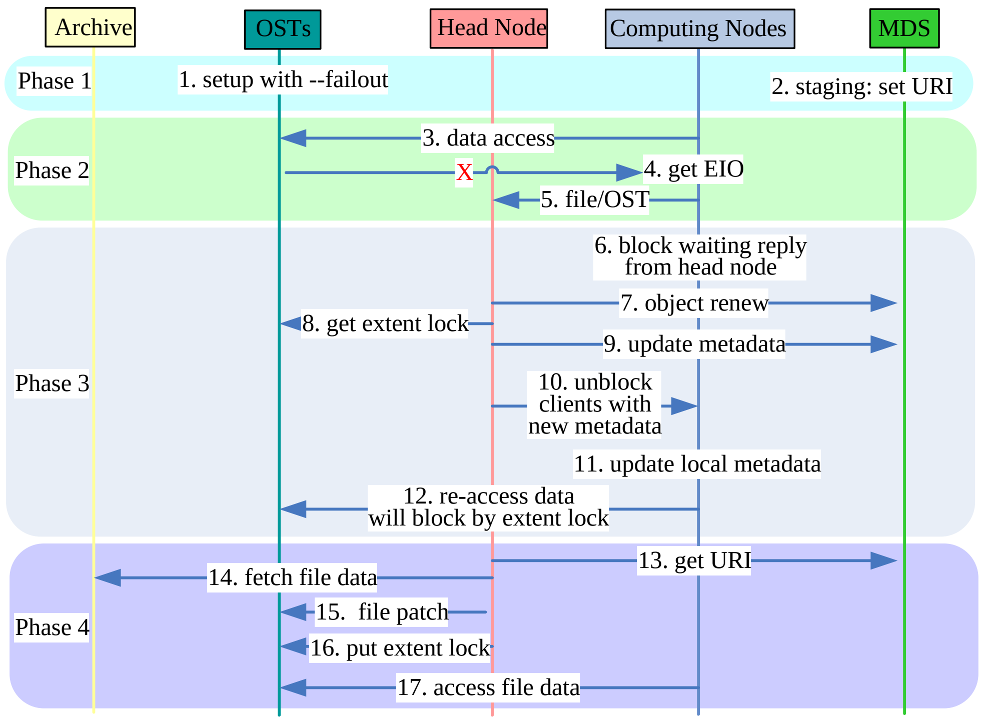


Figure 8: Steps for on-the-fly recovery

3.3.1 Phase 1: PFS Configuration and Metadata Setup

For on-the-fly recovery, the client needs to capture the OST failure case immediately. Hence, we configure all OSTs in Lustre's "fail-out" mode (step 1 of Figure 8). Thus, any operation referencing a file with a data stripe on a failed OST results in an immediate I/O error without ever blocking. In step 2, we further extend the metadata of the input files (at the MDS) with recovery information indicating the URI of a file's original source upon staging (see [44]).

3.3.2 Phase 2: Storage Failure Detection at Compute Nodes

To access the data of a file stored in the OST, the application issues calls *via* the standard POSIX file system API. The POSIX API is intercepted by the Lustre patched VFS system calls.

Due to the fail-out mode, both I/O node and data disk failures will lead to an immediate I/O error at the client upon file access (steps 3 and 4). By capturing the I/O error in the system function, we obtain file name and index of the failed OST or, in case of a disk failure, the location of the affected OST. In step 5, the client sends relevant information (file name, OST index) to the head node, which, in turn, initiates the data reconstruction. Hence, we perform online/real-time failure detection at the client for on-the-fly recovery during application execution, much in contrast to prior work on offline recovery that dealt with data loss prior to job activation [44].

3.3.3 Phase 3: Synchronization between Compute and Head Nodes

Upon receiving the data reconstruction request from the client, the head node performs two major tasks. First, it sends a request to the MDS, which locates a spare OST to replace the failed one and creates a new object for the file data on this spare. It next fetches the partial file data from the data source and populates the new object on the spare OST with it. When multiple compute nodes (Lustre clients) access the same data of this file, the head node only issues one reconstruction request per file per OST (even if multiple requests were received). At this point, compute nodes cannot access the object on the new OST as the data has not been populated. Once a stripe becomes available, compute nodes may access them immediately. To support such semantics, synchronization between the clients and OSTs is required. The fundamental mechanism for such synchronization is provided by Lustre locks.

Lustre Intent/Extent Lock Basics: Lustre provides two levels of locking, namely intent and extent locks. Intent locks arbitrate metadata requests from clients to MDS. Extent locks protect file operations on actual file data. Before modifying a file, an extent lock must be acquired. Each OST accommodates a lock server managing locks for stripes of data residing on that OST.

Synchronization Mechanism: We have implemented a centralized coordinator, a daemon residing on the head node. It consists of multiple threads that handle requests from clients and perform recovery. Upon arrival of a new request, the daemon launches the recovery procedure while the client remains blocked, just as other clients requesting data from this file/OST (step 6). Data recovery (step 7) is initiated by a novel addition to Lustre, the (*lfs objectrenew*) command. In response, the MDS locates a spare OST (on which the file does not reside yet) and creates a new object to replace the old one. Note that the MDS will not update its metadata information at this time. Instead, the update is deferred lazily to step 9 to allow accesses to proceed if they do not concern the failed OST.

In step 8, the daemon acquires the extent lock for the stripes of the new object. Since the (new) object information is hidden from other clients, there cannot be any contention for the lock. In step 9, the metadata information is updated, which utilizes the intent mechanism provided by Lustre again. In step 10, clients waiting for the patched data are unblocked and the new metadata is piggybacked. After clients update their locally cached metadata (step 11), they may already reference the new object. However, any access to the new object will still be blocked (step 12), this time due to their attempt to acquire the extent lock, which is still being held by the daemon on the head node.

Adjustment of the OST Extent Lock Grant Policy: In step 8, the daemon requests extent locks for all stripes of the recovery object. Consider the example in Fig. 9. Extent locks for stripes 2, 6, 10 and 14 are requested from OST 5. Upon a request for stripe 2, OST 5 grants the largest possible extent ([0,-1] where -1 denotes ∞) to the daemon. Afterward, requests for stripes 6, 10 and 14 match with lock [0,-1] resulting in an incremented reference count of the lock at the client without communicating with OST 5.

Our design modifies this default behavior of coarse-granular locking. We want to ensure that the extent lock to the stripes will be released one-by-one immediately after the respective stripe is patched. However, with Lustre distributed lock manager (DLM), the daemon only decrements the reference count on lock [0, -1] and releases it after all the stripes are patched.

To address this shortcoming, we adjust the extent lock grant policy at the OST server. Instead of granting the lock of [0,-1], a request from the daemon on the head node is granted only the exact range of stripes requested. This way, extent locks for different stripes differ (in step 8). Also, once a stripe is patched, the respective lock can be released so that other clients can access the patched data right away. Meanwhile, clients blocked on other stripes to be patched remain blocked on the extent locks. The extent lock policy is only updated for requests from the daemon on the head node without impacting the requests from other clients. Thus, it imposes no penalty in the non-failure case.

Such *metadata update delay* and *two-phase blocking of clients* provides the following properties: 1) Before any metadata update, clients can either access their cached data (which is consistent since stagein data is immutable) or request recovery (upon an I/O error). Either way, clients may still access the stripes of the old objects, but

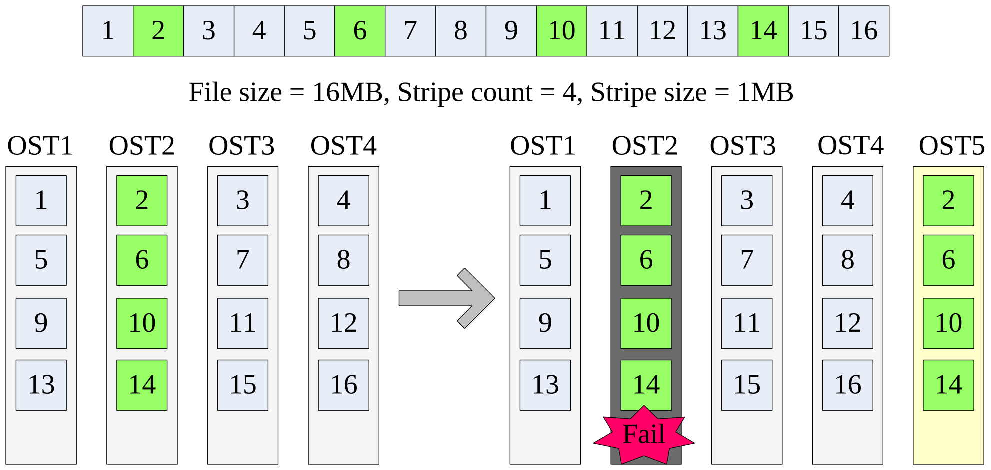
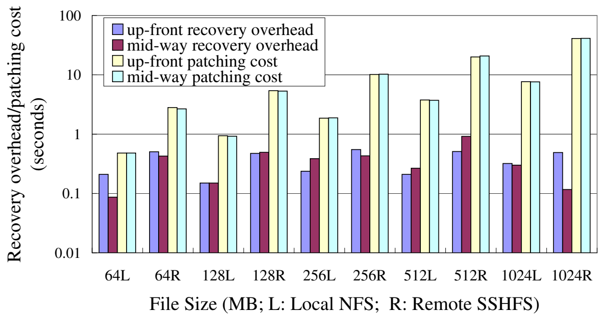
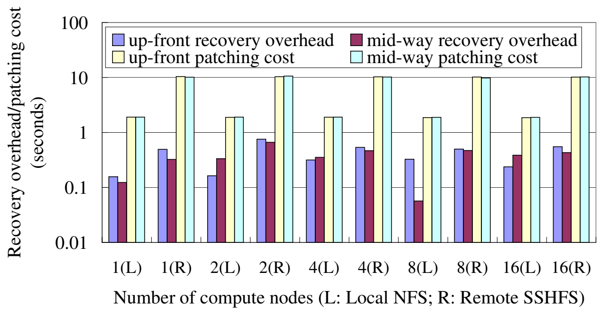


Figure 9: File reconstruction

the new objects remain invisible to them until the head node has patched the data and notified the clients to update the metadata. 2) Before patching the actual file data, the head node obtains an extent lock for all stripes of the new object, thereby blocking other clients that access the data now or later. 3) After patching the data, the extent locks per stripe are immediately released so that other clients can access partial data (stripes). Meanwhile, the daemon continues to patch subsequent stripes to provide pipelined overlap between patching and application progress. 4) The extent lock is further utilized for the second phase of blocking. Thus, data patching becomes an independent task that can be offloaded to the OSSs to distribute the patching workload in a scalable manner. 5) An OSS failure only affects a subset of the computing nodes (the Lustre clients) even though all the clients participate in the parallel I/O operations. Furthermore, most of the affected clients are blocked by the extent locks without any communication with the centralized coordinator on the head node, as discussed previously. Hence, the approach scales as communication with the centralized coordinator is limited to few nodes.



(a) Varied file size



(b) Varied # compute nodes

Figure 10: Matrix multiplication recovery overhead

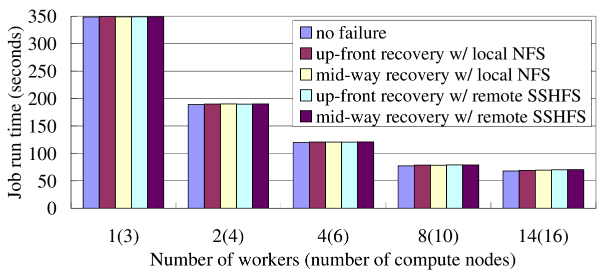
3.3.4 Phase 4: Data Reconstruction

In step 13, the URI of the remote file is obtained. In steps 14 and 15, stripes on the new object are populated. Due to per-stripe extent locks, stripes may be patched in any order. In our implementation, the clients subjected to I/O errors will supply the file range to access in their reconstruction request to the head node. The head node retains the order of the stripe requests and patches them accordingly. This speeds up application progress during reconstruction, particularly when files are accessed sequentially and a failure occurs in the middle of reading a file. In contrast, request-ignorant patching would hamper application progress by initiating a patch starting with the lowest indexed stripe of an OST, even though this stripe has already been read by clients.

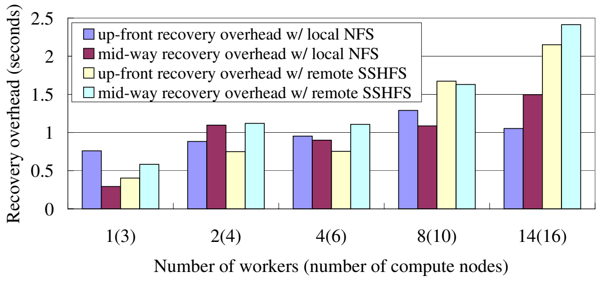
To this end, we have implemented a new Lustre command, *lfs patch*. Since phase 3 already obtains the extent lock for all the stripes, the new command can update the data range directly. Also, we set the file position in the patch system function instead of invoking lseek() at the user level. This allows us to bypass the overhead associated with automatic read-ahead (due to VFS caching). The extent lock for each stripe is released immediately after patching so that clients can access the stripe instantly (step 16).

3.4 Experimental Framework

We used the same 17-node linux cluster at NCSU as our testbed. The OS on each node was Fedora Core 5 Linux x86_64 with a Lustre-patched RHEL5 2.6.18 Linux kernel (Lustre 1.6.3). In our experiments, the cluster nodes were setup as I/O servers, compute nodes (Lustre clients), or both, as indicated below. We used different data staging sources for the job input data: (1) "/home" on the local NFS file system at the same HPC center with patching cost at 34.41MB/sec; (2) a server at another campus accessed by a file system client, SSHFS, based on Filesystems in Userspace (FUSE) and secure shell with a patching cost of 6.31MB/sec. Other patching sources, *e.g.*, GridFTP servers, might incur further delay. However, since most of the patching cost is shown to be overlapped with computation or I/O operations, changes in patching cost remain largely hidden from applications.



(a) Job run time



(b) Recovery overhead

Figure 11: mpiBLAST performance

3.5 Experimental Results

We assessed overhead and patching cost of on-the-fly recovery using an MPI benchmark and an MPI application.

3.5.1 Performance of Matrix Multiplication

We first assessed an MPI kernel that performs dense matrix multiplication (MM) with the standard $C = A \times B$ matrix operations, where A, B and C are $n \times n$ matrices. A and B are stored consecutively in an input file. We vary n to manipulate the size of the input file. Only one MPI task (the master) reads the input file before broadcasting the data to all the other tasks (workers). The matrix product $A \times B$ is distributed to all MPI processes. Since input occurs early during execution and since the code is more compute intensive, we focus on the recovery overhead, *i.e.*, the difference in job execution time of the jobs with and without failure.

Figure 10(a) shows the experimental results of matrix multiplication for increasing matrix dimensions, n (totaling 64MB, 128MB, 256MB, 512MB and 1GB). The MPI job runs on 16 compute nodes (one MPI task each). Figure 10(b) depicts the experimental results for varying number of compute nodes (1, 2, 4, 8 and 16) and a 256MB data input. For both of these tests, the *stripe count* (stripe width) for the input file was 4 and the *stripe size* was 1MB. We configured 5 OSTs (1OST/OSS) with the file residing on 4 OSTs and the spare OST for reconstruction. Some nodes double as both I/O and compute nodes. Since the configuration is the same, both with or without our solution, this provides a fair test environment.

To assess our system's capability to handle random storage failures, we varied the point in time where a failure occurred. In one experiment, we failed one of the OSTs up front, right as the MPI job started to run. This resulted in the master MPI task to experience an I/O error upon its first data access to the failed OST. In another experiment, we failed one OST mid-way during job execution. The master captures the I/O error immediately and sends a recovery request for the lost data to the daemon on the head node. Figures 10(a) and 10(b) indicate that the recovery overhead, from an application standpoint, is below 0.8 seconds for all cases. This is consistent in the sense that patching is overlapped with job I/O and hidden from the application. However, the actual time overlap between the patching and the job I/O varies. The recovery overhead for both up-front and mid-way recovery ranges from 0.06 to 0.75 seconds. Although the reconstruction cost in Figure 10(a) rises with file size, this is hidden from the application. While the patching cost from remote SSHFS is ~ 5 times that of local NFS, the recovery overhead for jobs patching from remote SSHFS is only slightly higher than local patching. The increase is dominated by the patching of the first stripe, which cannot be overlapped; subsequent stripes incur little extra cost.

3.5.2 Performance of mpiBLAST

We also assessed the performance of our solution using the mpiBLAST benchmark, a parallel implementation of NCBI BLAST, which splits a database into fragments and distributes the query tasks to workers by query segmentation before the BLAST search is performed in parallel.

Since mpiBLAST is more input-intensive, we discuss the impact of failure on the overall performance. Figure 11(a) shows the job run time. Figure 11(b) depicts the recovery overhead. mpiBLAST assigns one process to perform file output and another to schedule search tasks. Hence, the number of actual workers is the number of all the MPI processes minus two. Each worker accesses several files.

We configured 9 OSTs and increased compute nodes from 3 to 16 so that some double as server nodes (since our testbed has a total of 17 nodes). We distributed each input file to four of the OSTs by the Lustre stripe distribution policy and then failed one OST. As the number of worker processes increases, more files need to be accessed, *i.e.*, more files reside on the failed OST and require recovery so that the recovery overhead also increases (see Figure 11(b)). The number of failed files grows at the same rate as the workers. Compared to the overall runtime, the increase in recovery overhead is moderate. This is due to (1) parallel recovery of failed files referenced by disjoint workers and (2) reduced per-file patching cost for more workers as file sizes decrease due to work sharing. Figure 11(b) shows that the recovery overhead for jobs patching from remote SSHFS is higher than for local patching due to the slower data source. Also, with more workers, more failed files exist. Consequently, recovery becomes more costly, yet at a moderate growth rate due to the aforementioned overlap. For the benchmarks we used, such moderate recovery overhead is negligible compared with the job runtime. We expect that the same holds true for most supercomputing jobs as large jobs tend to run much longer and as input files are typically only read in the job initialization phase. Wallclock time estimates generally cover such negligible overhead. Hence, additional time need not be budgeted for the job due to our techniques.

3.6 Conclusion

We have presented the design of a novel on-the-fly recovery framework as a means to address fault tolerance within parallel file systems in HPC centers. The recovery framework provides a seamless way for a running job's input data to be reconstructed from its remote source in case of I/O errors. We have designed the system to take advantage of key characteristics of HPC I/O workloads such as their immutable input data, sequential access and persistent remote copy. We have further implemented this design into the Lustre parallel file system commonly used in supercomputer centers. Results with I/O-intensive MPI benchmarks suggest that the recovery mechanism imposes little overhead. Both HPC centers and users stand to benefit from improved serviceability, data availability and reduced job turnaround time in the face of storage system failure.

4 Temporal Replication of Job Input Data

4.1 Introduction

Currently, the majority of disk failures are masked by hardware solutions such as RAID [30]. However, it is becoming increasingly difficult for common RAID configurations to hide disk failures as disk capacity is expected to grow by 50% each year, which increases the reconstruction time. The reconstruction time is further prolonged by the "polite" policy adopted by RAID systems to make reconstruction yield to application requests. This causes a RAID group to be more vulnerable to additional disk failures during reconstruction [35].

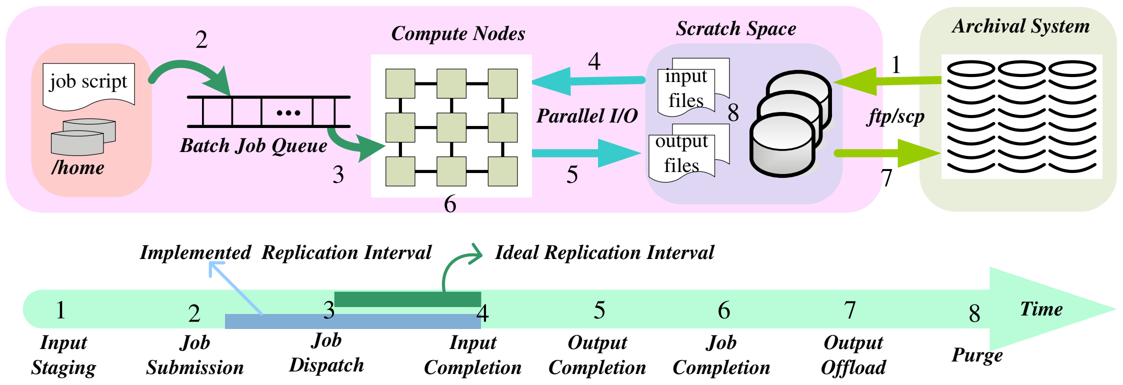


Figure 12: Event timeline with ideal and implemented replication intervals

According to recent studies [23], disk failures are only part of the sources causing data unavailability in storage systems. RAID cannot help with storage node failures. In next-generation supercomputers, thousands or even tens of thousands of I/O nodes will be deployed and will be expected to endure multiple concurrent node failures at any given time. Consider the Jaguar system at Oak Ridge National Laboratory, which is on the roadmap to a petaflop machine (currently No. 5 on the Top500 list with 23,412 cores and hundreds of I/O nodes). Our experience with Jaguar shows that the majority of whole-system shutdowns are caused by I/O nodes' software failures. Although parallel file systems, such as Lustre [13], provide storage node failover mechanisms, our experience with Jaguar again shows that this configuration might conflict with other system settings. Further, many supercomputing centers hesitate to spend their operations budget on replicating I/O servers and instead of purchasing more FLOPS.

Figure 12 gives an overview of an event timeline describing a typical supercomputing job's data life-cycle. Users stage their job input data from elsewhere to the scratch space, submit their jobs using a batch script, and offload the output files to archival systems or local clusters. For better space utilization, the scratch space does not enforce quotas but purges files after a number of days since the last access. Moreover, job input files are often read-only (also read-once) and output files are write-once.

Although most supercomputing jobs performing numerical simulations are output-intensive rather than input-intensive, the input data availability problem poses two unique issues. First, input operations are more sensitive to server failures. Output data can be easily redirected to survive runtime storage failures using *eager offloading* [29]. As mentioned earlier, many systems like Jaguar do not have file system server failover configurations to protect against input data unavailability. In contrast, during the output process, parallel file systems can more easily skip failed servers in striping a new file or perform restriping if necessary. Second, loss of input data often brings heavier penalty. Output files already written can typically withstand temporary I/O server failures or RAID reconstruction delays as job owners have days to perform their stage-out task before the files are purged from the scratch space. Input data unavailability, on the other hand, incurs job termination and resubmission. This introduces high costs for job re-queuing, typically orders of magnitude larger than the input I/O time itself.

Fortunately, unlike general-purpose systems, in supercomputers we can anticipate *future* data accesses by checking the job scheduling status. For example, a compute job is only able to read its input data during its execution. By coordinating with the job scheduler, a supercomputer storage system can selectively provide additional protection only for the duration when the job data is expected to be accessed.

We proposed temporal file replication, wherein a parallel file system performs transparent and temporary replication of job input data. This facilitates fast and easy file reconstruction before and during a job's execution without additional user hints or application modifications. Unlike traditional file replication techniques, which have mainly been designed to improve long-term data persistence and access bandwidth or to lower access latency, the temporal replication scheme targets the enhancement of short-term data availability centered around job executions in supercomputers.

We have implemented our scheme in the popular Lustre parallel file system and combined it with the Moab job scheduler by building on our previous work on coinciding input data staging alongside computation [44]. We have also implemented a replication-triggering algorithm that coordinates with the job scheduler to delay the replica

Table 2: Configurations of top five supercomputers as of 06/2008

System	# Cores	Aggr-egate Memory (TB)	Scratch Space (TB)	Memory to Storage Ratio	Top 500 Rank
RoadRunner(LANL)	122400	98	2048	4.8%	1
BlueGene/L(LLNL)	106496	73.7	1900	3.8%	2
BlueGene/P(Argonne)	163840	80	1126	7.1%	3
Ranger(TACC)	62976	123	1802	6.8%	4
Jaguar(ORNL)	23412	46.8	600	7.8%	5

creation. Using this approach, we ensure that the replication completes in time to have an extra copy of the job input data before its execution.

We then evaluate the performance by conducting real-cluster experiments that assess the overhead and scalability of the replication-based data recovery process. Our experiments indicate that replication and data recovery can be performed quite efficiently. Thus, our approach presents a novel way to bridge the gap between parallel file systems and job schedulers, thereby enabling us to strike a balance between an HPC center resource consumption and serviceability.

4.2 Temporal Replication Design

Supercomputers are heavily utilized. Most jobs spend significantly more time waiting in the batch queue than actually executing. The popularity of a new system ramps up as it goes towards its prime time. For example, from the 3-year Jaguar job logs, the average job wait-time:run-time ratio increases from 0.94 in 2005, to 2.86 in 2006, and 3.84 in 2007.

4.2.1 Justification and Design Rationale

A key concern about the feasibility of temporal replication is the potential space and I/O overhead replication might incur. However, we argue that by replicating selected "active files" during their "active periods", we are only replicating a small fraction of the files residing in the scratch space at any given time. To estimate the extra space requirement, we examined the sizes of the aggregate memory space and the scratch space on state-of-the-art supercomputers. The premise is that with today's massively parallel machines and with the increasing performance gap between memory and disk accesses, batch applications are seldom out-of-core. This also agrees with our observed memory use pattern on Jaguar (see below). Parallel codes typically perform input at the beginning of a run to initialize the simulation or to read in databases for parallel queries. Therefore, the aggregate memory size gives a bound for the total input data size of active jobs. By comparing this estimate with the scratch space size, we can assess the relative overhead of temporal replication.

Table 2 summarizes such information for the top five supercomputers [1]. We see that the memory-to-storage ratio is less than 8%. Detailed job logs with per-job peak memory usage indicate that the above approximation of using the aggregate memory size significantly overestimates the actual memory use (discussed later in this subsection). While the memory-to-storage ratio provides a rough estimation of the replication overhead, in reality, however, a number of other factors need to be considered. First, when analyzing the storage space overhead, queued jobs' input files cannot be ignored, since their aggregate size can be even larger than that of running jobs. In the following sections, we propose additional optimizations to shorten the lifespan of replicas. Second, when analyzing the bandwidth overhead, the frequency of replication should be taken into account. Jaguar's job logs show an average job run time of around 1000 seconds and an average aggregate memory usage of 31.8 GB, which leads to a bandwidth consumption of less than 0.1% of Jaguar's total capacity of 284 GB/s. For this reason, we primarily focus on the space overhead in the following discussions.

Next, we discuss a supercomputer's usage scenarios and configuration in more detail to justify the use of replication to improve job input data availability.

Even though replication is a widely used approach in many distributed file system implementations, it is seldom adopted in supercomputer storage systems. In fact, many popular high-performance parallel file systems (e.g., Lustre and PVFS) do not even support replication, mainly due to space concerns. The capacity of the scratch space is important in (1) allowing job files to remain for a reasonable amount of time (days rather than hours), avoiding the loss of precious job input/output data, and (2) allowing giant "hero" jobs to have enough space to generate their output. Blindly replicating all files, even just once, would reduce the effective scratch capacity to half of its original size.

Temporal replication addresses the above concern by leveraging job execution information from the batch scheduler. This allows it to only replicate a small fraction of "active files" in the scratch space by letting the "replication window" slide as jobs flow through the batch queue. Temporal replication is further motivated by several ongoing trends in supercomputer configurations and job behavior. First, as mentioned earlier, Table 2 shows that the memory to scratch space ratio of the top 5 supercomputers is relatively low. Second, it is rather rare for parallel jobs on these machines to fully consume the available physical memory on each node. A job may complete in shorter time on a larger number of nodes due to the division of workload and data, resulting in lower per-node memory requirements at a comparable time-node charge. Figure 13 shows the per-node memory usage of both *running* and *queued* jobs over one month on the ORNL Jaguar system. It backs our hypothesis that jobs tend to be in-core, with their aggregate peak memory usage providing an upper bound for their total input size. We also found the actual aggregate memory usage averaged over the 300 sample points to be significantly below the total amount of memory available shown in Table 2: 31.8 GB for running jobs and 49.5 GB for queued jobs.

4.2.2 Delayed Replica Creation

Based on the above observations about job wait times and cost/benefit trade-offs for replication in storage space, we propose the following design of an HPC-centric file replication mechanism.

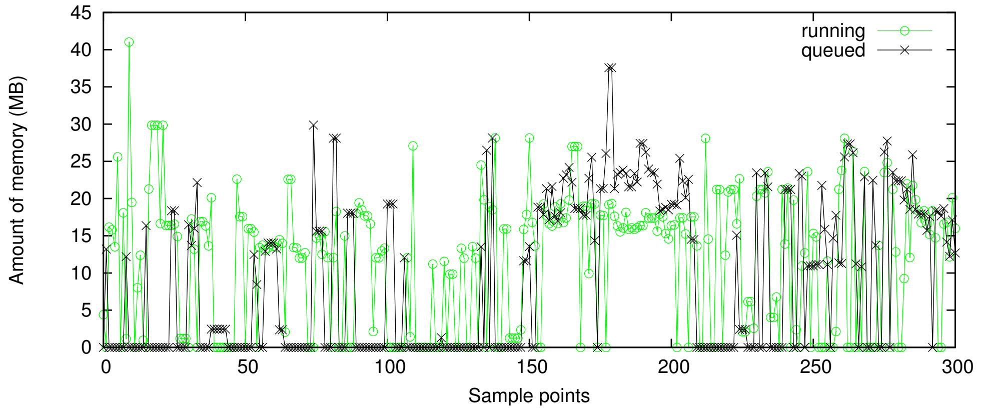


Figure 13: Per-node memory usage from 300 uniformly sampled time points over a 30-day period based on job logs from the ORNL Jaguar system. For each time point, the total memory usage is the sum of peak memory used by all jobs in question.

When jobs spend a significant amount of time waiting, replicating their input files (either at stage-in or submission time) wastes storage space. Instead, a parallel file system can obtain the current queue status and determine a *replication trigger point* to create replicas for a given job. The premise here is to have enough jobs near the top of the queue, stocked up with their replicas, such that jobs dispatched next will have extra input data redundancy. Additional replication will be triggered by job completion events, which usually result in the dispatch of one or more jobs from the queue. Since jobs are seldom interdependent, we expect that supplementing a modest prefix of the queued jobs with a second replica of their input will be sufficient. Only one copy of a job's input data will be available before its replication trigger point. However, this primary copy can be protected with periodic availability checks and remote data recovery techniques previously developed and deployed by us [44].

Completion of a large job is challenging as it can activate many waiting jobs requiring instant replication of multiple datasets. As a solution, we propose to query the queue status from the job scheduler. Let the replication window, w, be the length of the prefix of jobs at the head of the queue that should have their replicas ready. w should be the smallest integer such that:

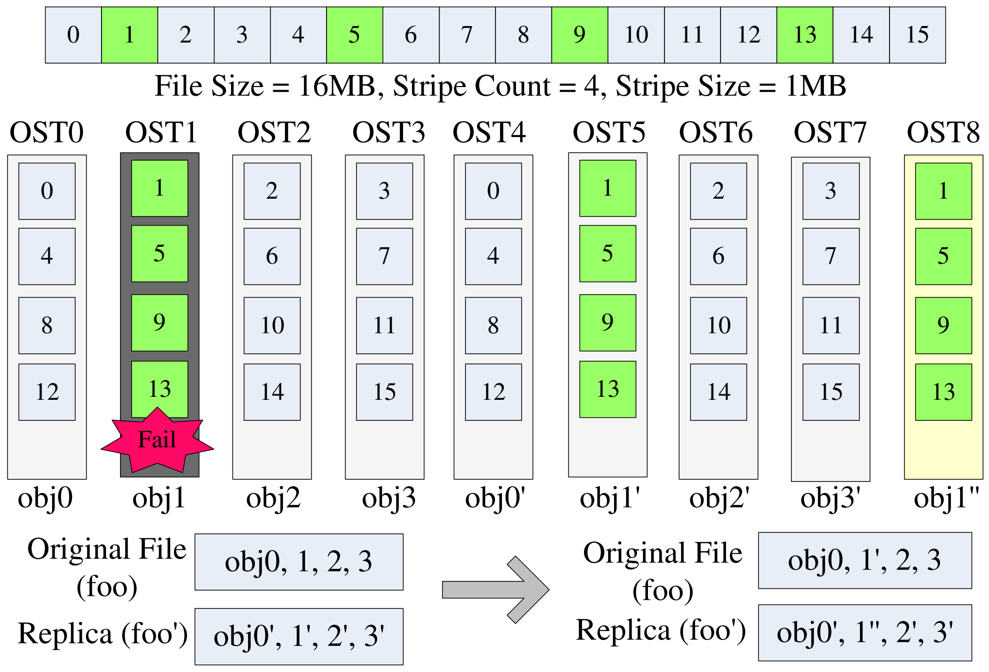


Figure 14: Objects of an original job input file and its replica. A failure occurred to OST1, which caused accesses to the affected object to be redirected to their replicas on OST5, with replica regeneration on OST8.

$$\sum_{i=0}^{w} |Q_i| > max(R, \alpha S),$$

where $|Q_i|$ is the number of nodes requested by the ith ranked job in the queue, R is the number of nodes used by the largest running job, S is the total number of nodes in the system, and the factor $\alpha(0 \leq \alpha)$ is a controllable parameter to determine the eagerness of replication.

One problem with the above approach is that job queues are quite dynamic as strategies such as backfilling are typically used with an FCFS or FCFS-with-priority scheduling policy. Therefore, jobs do not necessarily stay in the queue in their arrival order. In particular, jobs that require a small number of nodes are likely to move ahead faster. To address this, we augment the above replication window selection with a "shortcut" approach and define a threshold T, $0 \leq T \leq 1$. Jobs that request $T \cdot S$ nodes will have their input data replicated immediately regardless of the current replica window. This approach allows jobs that tend to be scheduled quickly to enjoy early replica creation.

4.2.3 Eager Replica Removal

We can also shorten the replicas' life span by removing the extra copy once we know it is not needed. A relatively safe approach is to perform the removal at job completion time. Although users sometimes submit additional jobs using the same input data, the primary data copy will again be protected with our offline availability check and recovery [44]. Further, subsequent jobs will also trigger replication as they progress toward the head of the job queue.

Overall, we recognize that the input files for most in-core parallel jobs are read at the beginning of job execution and never re-accessed thereafter. Hence, we have designed an *eager replica removal* strategy that removes the extra replica once the replicated file has been closed by the application. This significantly shortens the replication duration, especially for long-running jobs. Such an aggressive removal policy may subject input files to a higher risk in the rare case of a subsequent access further down in its execution. However, we argue that reduced space requirements for the more common case outweigh this risk.

4.3 Implementation Issues

All our modifications were made within Lustre and do not affect the POSIX file system APIs. Therefore, data replication, failover and recovery processes are entirely transparent to user applications.

In our implementation, a supercomputer's head node doubles as a replica management service node, running as a Lustre client. Job input data is usually staged via the head node making it well suited for initiating replication operations. Replica management involves generating a copy of the input dataset at the appropriate replication trigger point, scheduling periodic failure detection before job execution, and also scheduling data recovery in response to reconstruction requests. Data reconstruction requests are initiated by the compute nodes when they observe storage failures during file accesses. The replica manager serves as a coordinator that facilitates file reorganization, replica reconstruction, and streamlining of requests from the compute nodes in a non-redundant fashion.

Replica Creation and Management: We use the copy mechanism of the underlying file system to generate a replica of the original file. In creating the replica, we ensure that it inherits the striping pattern of the original file and is distributed on I/O nodes disjoint from the original file's I/O nodes. As depicted in Figure 14, the objects of the original file and the replica form pairs (objects $(0, 0')$, $(1, 1')$, etc.). The replica is associated with the original file for its lifetime by utilizing Lustre's extended attribute mechanism.

Failure Detection: For persistent data availability, we perform periodic failure detection before a job's execution. This offline failure detection mechanism was described in our previous work [44]. The same mechanism has been extended for transparent storage failure detection and access redirection during a job run. Both I/O node failures and disk failures will result in an I/O error immediately within our Lustre patched VFS system calls. Upon capturing the I/O error in the system function, Lustre obtains the file name and the index of the failed OST. Such information is then sent by the client to the head node, which, in turn, initiates the object reorganization and replica reconstruction procedures.

Object Failover and Replica Regeneration: Upon an I/O node failure, either detected by the periodic offline check or by a compute node through an I/O error, the aforementioned file and failure information is sent to the head node. Using several new commands that we have developed, the replica manager will query the MDS to identify the appropriate objects in the replica file that can be used to fill the holes in the original file. The original file's metadata is updated subsequently to integrate the replicated objects into the original file for seamless data access failover. Since metadata updates are inexpensive, the head node is not expected to become a potential bottleneck.

To maintain the desired data redundancy during the period that a file is replicated, we choose to create a "secondary replica" on another OST for the failover objects after a storage failure. The procedure begins by locating another OST, giving priority to one that currently does not store any part of the original or the primary replica file.[1] Then, the failover objects are copied to the chosen OST and in turn integrated into the primary replica file. Since the replica acts as a backup, it is not urgent to populate its data immediately. In our implementation, such stripe-wise replication is delayed by 5 seconds (tunable) and is offloaded to I/O nodes (OSSs).

Streamlining Replica Regeneration Requests: Due to parallel I/O , multiple compute nodes (Lustre clients) are likely to access a shared file concurrently. Therefore, in the case of a storage failure, we must ensure that the head node issues a single failover/regeneration request per file and per OST despite multiple such requests from different compute nodes. We have implemented a centralized coordinator inside the replica manager to handle the requests in a non-redundant fashion.

4.4 Experimental Results

To evaluate the temporal replication scheme, we performed real-cluster experiments. We assessed our implementation of temporal replication in the Lustre file system in terms of the online data recovery efficiency.

[1] In Lustre, file is striped across 4 OSTs by default. Since supercomputers typically have hundreds of OSTs, an OST can be easily found.

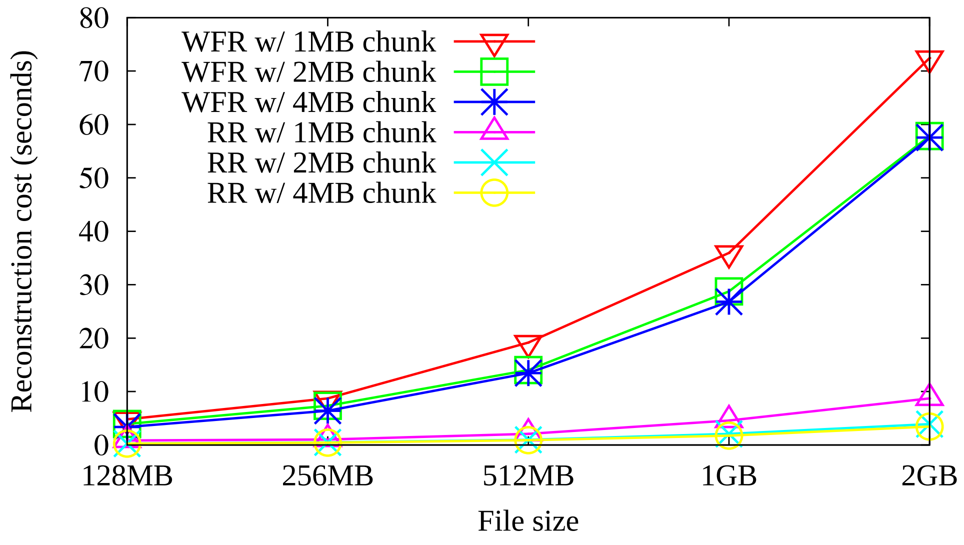


Figure 15: Offline replica reconstruction cost with varied file size

4.4.1 Failure Detection and Offline Recovery

Before a job begins to run, we periodically check for failures on OSTs that carry its input data. The detection cost is less than 0.1 seconds as the number of OSTs increases to 256 (16 OSTs on each of the 16 OSSs) in our testbed. Since failure detection is performed when a job is waiting, it incurs no overhead on job execution itself. When an OST failure is detected, two steps are performed to recover the file from its replica: object failover and replica reconstruction. The overhead of object failover is relatively constant (0.84-0.89 seconds) regardless of the number of OSTs and the file size. This is due to the fact that the operation only involves the MDS and the client that initiates the command. Figure 15 shows the replica reconstruction (RR) cost with different file sizes. The test setup consisted of 16 OSTs (1 OST/OSS). We varied the file size from 128MB to 2GB. With one OST failure, the data to recover ranges from 8MB to 128MB causing a linear increase in RR overhead. Figure 15 also shows that the *whole file reconstruction (WFR)*, the conventional alternative to our more selective scheme where the entire file is re-copied, has a much higher overhead. In addition, RR cost increases as the chunk size decreases due to the increased fragmentation of data accesses.

4.4.2 Online Recovery Application 1: Matrix Multiplication (MM)

To measure on-the-fly data recovery overhead during a job run with temporal replication, we used MM, an MPI kernel that performs dense matrix multiplication. It computes the standard $C = A * B$ operation, where A, B and C are $n * n$ matrices. A and B are stored contiguously in an input file. We vary n to manipulate the problem size. Like in many applications, only one master process reads the input file, then broadcasts the data to all the other processes for parallel multiplication using a BLOCK distribution.

Figure 16 depicts the MM recovery overhead with different problem sizes. Here, the MPI job ran on 16 compute nodes, each with one MPI process. The total input size was varied from 128MB to 2GB by adjusting n. We configured 9 OSTs (1 OST/OSS), with the original file residing on 4 OSTs, the replica on another 4, and the reconstruction of the failover object occurring on the remaining one. Limited by our cluster size, we let nodes double as both I/O and compute nodes.

To simulate random storage failures, we varied the point in time where a failure occurs. In "up-front", an OSTs failure was induced right before the MPI job started running. Hence, the master process experienced an I/O error upon its first data access to the failed OST. With "mid-way", one OST failure was induced mid-way during the input process. The master encountered the I/O error amidst its reading and sent a recovery request to the replica manager on the head node. Figure 16 indicates that the application-visible recovery overhead was almost constant for all cases (right around 1 second) considering system variances. This occurs because only one

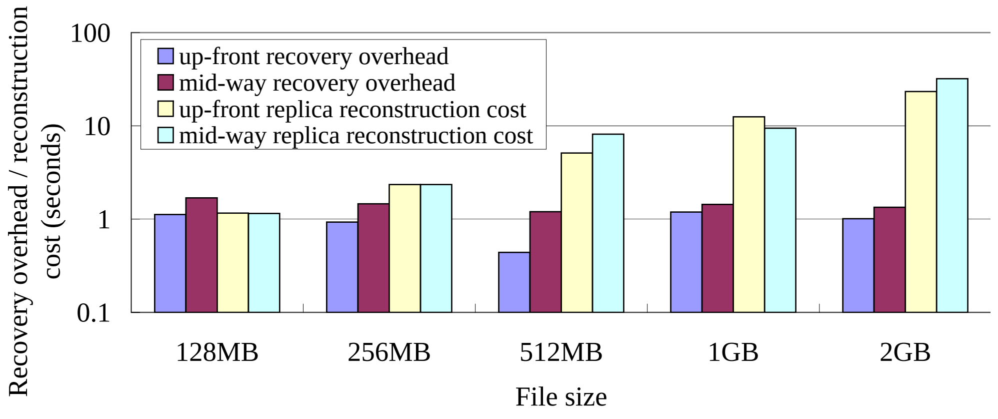


Figure 16: MM recovery overhead vs. replica reconstruction cost

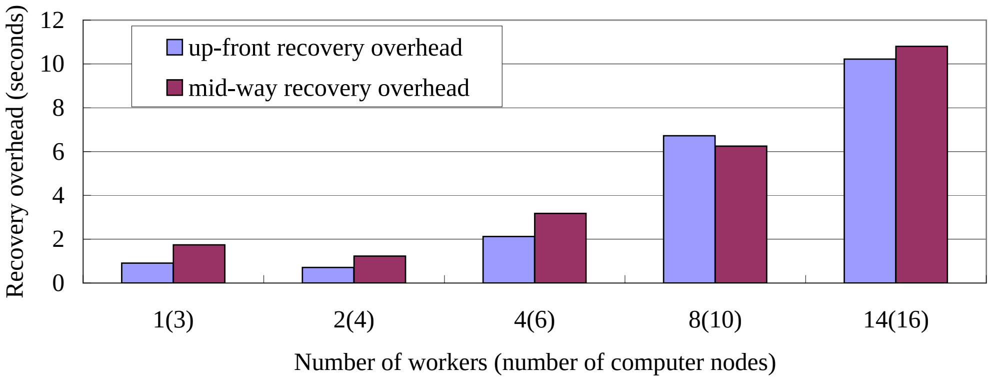


Figure 17: Recovery overhead of mpiBLAST

object was replaced for all test cases while only one process was engaged in input. Even though the replication reconstruction cost rises as the file size increases, this was hidden from the application. The application simply progressed with the failover object from the replica while the replica itself was replenished in the background.

4.4.3 Online Recovery Application 2: mpiBLAST

To evaluate the data recovery overhead using temporal replication with a read-intensive application, we tested with mpiBLAST [16], which splits a database into fragments and performs a BLAST search on the worker nodes in parallel. Since mpiBLAST is more input-intensive, we examined the impact of a storage failure on its overall performance. The difference between the job execution times with and without failure, i.e., the recovery overhead, is shown in Figure 17. Since mpiBLAST assigns one process as the master and another to perform file output, the number of actual worker processes performing parallel input is the total process number minus two.

The Lustre configurations and failure modes used in the tests were similar to those in the MM tests. Overall, the impact of data recovery on the application's performance was small. As the number of workers grew, the database was partitioned into more files. Hence, more files resided on the failed OST and needed recovery. As shown by Figure 17, the recovery overhead grew with the number of workers. Since each worker process performed input at its own pace and the input files were randomly distributed to the OSTs, the I/O errors captured on the worker processes occurred at different times. Hence, the respective recovery requests to the head node were not issued synchronously in parallel but rather in a staged fashion. With many applications that access a fixed number of shared input files, we expect to see a much more scalable recovery cost with regard to the number of MPI processes using our techniques.

4.5 Conclusion

We have presented a novel temporal replication scheme for supercomputer job data. By creating additional data redundancy for transient job input data, we allow fast online data recovery from local replicas without user intervention or hardware support. This general-purpose, high-level data replication can help avoid job failures/resubmission by reducing the impact of both disk failures or software/hardware failures on the storage nodes. Our implementation, using the widely used Lustre parallel file system and the Moab scheduler, demonstrates that replication and data recovery can be performed efficiently.

5 Related Work

RAID Recovery: Disk failures can often be masked by standard RAID techniques [30]. However, RAID is geared toward whole disk failures and does not address sector-level faults [5, 20, 33]. It is further impaired by controller failures and multiple disk failures within the same group. Without hot spares, reconstruction requires manual intervention and is time consuming. With RAID reconstruction, disk arrays either run in a degraded (not yielding to other I/O requests) or polite mode. In a degraded mode, busy disk arrays suffer a substantial performance hit when crippled with multiple failed disks [43, 38]. This degradation is even more significant on parallel file systems as files are striped over multiple disk arrays and large sequential accesses are common. Under a polite mode, with rapidly growing disk capacity, the total reconstruction time is projected to increase to days subjecting a disk array to additional failures [35]. Furthermore, RAID technology cannot handle a media error, also know as an unrecoverable read error (URE), which occurs during dailed disk reconstruction [3]. With 50 GB SATA disk drives, MTBF is about one error every 10^{14} bits (12.5 terabytes), so that a media error was very unlikely to occur. When it does occur during reconstruction, a 50 GB disk takes only a few hours to recover from tape. However, for terabyte disks, the disk failure plus media error scenario becomes almost inevitable. Recovering the storage array from backup tape could take a month. To address this problem, Panasas has implemented a significant extension, called “tiered parity” [4], to RAID. In this model, Panasas has built “vertical parity” and “network parity” on top of existing “horizontal parity”. With vertical parity, they have added RAID within each disk. According to Panasas, vertical parity reduces the error rate to between one in 10^{18} and one in 10^{19} bits written, which is 1000 to 10,000 times better than the URE rate. The extra parity information uses 10 percent of the disk capacity. On top of horizonal and vertical parity schemes, Panasas also adds an additional layer of network parity protection. At this level, parity checking is done on the client side. Our approach complements RAID systems by providing fast recovery protecting against non-disk and multiple disk failures.

Recent work on popularity-based RAID reconstruction [39] rebuilds more frequently accessed data first, thereby reducing reconstruction time and user-perceived penalties. However, supercomputer storage systems host transient job data, where “unaccessed” job input files are often more important than “accessed” ones. In addition, such optimizations cannot cope with failures beyond RAID's protection at the hardware level.

Replication: Data replication, a commonly used technique for persistent data availability, creates and stores redundant copies (*replicas*) of datasets. Various replication techniques have been studied [9, 14, 37, 41] in many distributed file systems [10, 17, 26]. Most existing replication techniques treat all datasets with equal importance and each dataset with static, time-invariant importance when making replication decisions. An intuitive improvement would be to treat datasets with different priorities. To this end, BAD-FS [6] performs selective replication according to a cost-benefit analysis based on the replication costs and the system failure rate. Similar to BAD-FS, our temporal replication approached also makes on-demand replication decisions. However, our scheme is more “access-aware” rather than “cost-aware”. While BAD-FS still creates static replicas, our replication approach utilizes explicit information from the job scheduler to closely synchronize and limit replication to jobs in execution or soon to be executed.

Parallel File Systems: There are two classes of parallel file systems, namely those whose architectures are based on I/O nodes/servers managing data on directly attached storage devices (such as PVFS [12] and LUSTRE [13]) and those with centralized, shared storage devices that are shared by all I/O nodes (such as GPFS [34]). For the former category, node failure implies that a partition of the storage system is unavailable. Since parallel file systems usually stripe datasets for better I/O performance, failure of one node may affect a large portion of user jobs. Moreover, unlike specialized nodes/servers such as metadata servers, token servers, etc., I/O nodes in parallel file systems may not be routinely protected through failover. I/O node failover does not help when the underlying RAID recovery is impaired (as mentioned above) as the data is seldom replicated. Our offline and online recovery solution for job input data mitigates these situations by exploiting the source copies of user job input data.

I/O shepherding [20] introduces a reliability infrastructure for file systems by executing I/O requests using user-specified FT mechanisms including retries, sanity checking, checksums, and mirrors or parity protection to recover from lost blocks or disks. This work is similar in the sense that it attempts to introduce fault-tolerant behavior into file systems by reliably executing I/O requests. However, we are concerned with HPC job input data and rely on external sources for I/O node failure recovery.

Erasure Coding: Another widely investigated technique is erasure coding [11, 32, 42]. With erasure coding, k parity blocks are encoded into n blocks of source data. When a failure occurs, the whole set of $n + k$ blocks of data can be reconstructed with any n surviving blocks through decoding.

Erasure coding reduces the space usage of replication but adds computational overhead for data encoding/decoding. In [40], the authors provide a theoretical comparison between replication and erasure coding. In many systems, erasure coding provides better overall performance balancing computation costs and space usage. However, for supercomputer centers, its computation costs will be a concern. This is because computing time in supercomputers is a precious commodity. At the same time, our data analysis suggests that the amount of storage space required to replicate data for active jobs is relatively small compared to the total storage footprint. Therefore, compared to erasure coding, our replication approach is more suitable for supercomputing environments, which is verified by our experimental study.

6 Conclusion

In the area of FT, techniques for job input data preservation, remote patching and temporal replication were developed to improve the reliability, availability and performance of HPC I/O systems. We investigate offline and online approaches for reconstructing missing pieces of datasets from data sources where the job input data was originally staged from. The two approaches complement each other. We have shown that supercomputing centers' data availability can be drastically enhanced by periodically checking and reconstructing datasets for queued jobs while the reconstruction overheads are barely visible to users. The approach provides a seamless way for a running job's input data to be reconstructed from its remote source in case of I/O errors. We have presented a novel temporal replication scheme by creating additional data redundancy for transient job input data to avoid job failures/resubmission.

Both remote patching and temporal replication will be able to help with storage failures at multiple layers. While remote patching poses no additional space overhead, the patching costs depend on the data source and the end-to-end network transfer performance. It can be hidden from applications during a job's execution. Temporal replication, on the other hand, trades space (which is relatively cheap at supercomputers) for performance. It provides high-speed data recovery and reduces the space overhead by only replicating the data when it is needed.

Experimental results with I/O-intensive MPI benchmarks indicate that the recovery mechanism of the file remote patching imposes little overhead and scales well with increasing file sizes. This includes three independent steps: checking all the OSTs in parallel, updating file stripe metadata and reconstructing the missing data. Furthermore, online remote patching can be overlapped with computation and communication as data stripes are recovered. Our experimental results reinforce this by showing that the increase in job execution time due to on-the-fly recovery is negligible compared to non-faulting runs. Experimental results also indicate the impact of data recovery from temporal replication on the application's performance is small since applications simply progress with failover objects from the replicas, while replicas themselves are replenished in the background.

Overall, the offline/online recovery and temporal replication approaches for job input data tolerate a majority of the failures with storage and I/O.

Acknowledgments

This research was supported in part by, and used the resources of, the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at ORNL, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725).

This work was supported in part by NSF grants CNS-1058779, HECURA-0937908, CNS-0958311 and CCF-0937690.

References

[1] Top 500 list. http://www.top500.org/, June 2002.

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

[3] Panasas Invents 'Tiered Parity'. feature, HPCwire, 2007.

[4] Panasas tiered parity architecture. white paper, Panasas, 2008.

[5] L. Bairavasundaram, G. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *SIGMETRICS*, pages 289 – 300, June 2007.

[6] J. Bent, D. Thain, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Explicit control in a batch aware distributed file system. In *Proceedings of the First USENIX/ACM Conference on Networked Systems Design and Implementation*, Mar. 2004.

[7] 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.

[8] 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 6th Workshop on Input/Output in Parallel and Distributed Systems*, 1999.

[9] C. Blake and R. Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *Proceedings the 9th Workshop on Hot Topics in Operating Systems (HotOS)*, 2003.

[10] 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.

[11] 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.

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

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

[14] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proceedings of the ACM SIGCOMM Conference*, 2002.

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

[16] A. E. Darling, L. Carey, and W. chun Feng. The design, implementation, and evaluation of mpiblast. In *ClusterWorld Conference & Expo and the 4th International Conference on Linux Cluster: The HPC Revolution '03*, June 2003.

[17] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Symposium on Operating Systems Principles*, 2003.

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

[19] 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.

[20] H. Gunawi, V. Prabhakaran, S. Krishnan, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Improving file system reliability with i/o shepherding. In *Symposium on Operating Systems Principles*, Oct. 2007.

[21] C. Hsu and W. Feng. A power-aware run-time system for high-performance computing. In *SC*, 2005.

[22] C.-H. Hsu and W.-C. Feng. A power-aware run-time system for high-performance computing. In *Supercomputing*, 2005.

[23] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *Trans. Storage*, 4(3):1–25, 2008.

[24] O. R. N. Laboratory. Resources - national center for computational sciences (nccs). http://info.nccs.gov/resources/jaguar, June 2007.

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

[26] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 226–38. Association for Computing Machinery SIGOPS, 1991.

[27] X. Ma, S. Vazhkudai, V. Freeh, T. Simon, T. Yang, and S. L. Scott. Coupling prefix caching and collective downloads for remote data access. In *Proceedings of the ACM International Conference on Supercomputing*, 2006.

[28] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, 2003.

[29] H. Monti, A. Butt, and S. S. Vazhkudai. Timely Offloading of Result-Data in HPC Centers. In *Proceedings of 22nd Int'l Conference on Supercomputing* $ICS'08$, June 2008.

[30] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, 1988.

[31] I. Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of HPCA-11*. IEEE Computer Society, 2005.

[32] J. Plank, A. Buchsbaum, R. Collins, and M. Thomason. Small parity-check erasure codes - exploration and observations. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2005.

[33] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. G. abd Andrea C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Symposium on Operating Systems Principles*, pages 206 – 220, Oct. 2005.

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

[35] B. Schroeder and G. Gibson. Understanding failure in petascale computers. In *SciDAC Conference*, 2007.

[36] S. C. Simms, G. G. Pike, and D. Balog. Wide area filesystem performance using lustre on the teragrid. In *TeraGrid*, 2007.

[37] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference*, 2001.

[38] A. Thomasian, G. Fu, and C. Han. Performance of two-disk failure-tolerant disk arrays. *IEEE Transactions on Computers*, 56(6):799–814, 2007.

[39] L. Tian, D. Feng, H. Jiang, K. Zhou, L. Zeng, J. Chen, Z. Wang, and Z. Song. Pro: a popularity-based multi-threaded reconstruction optimization for raid-structured storage systems. In *USENIX Conference on File and Storage Technologies*, pages 32–32, Berkeley, CA, USA, 2007. USENIX Association.

[40] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, 2002.

[41] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Operating Systems Design and Implementation*, Nov. 2006.

[42] J. J. Wylie and R. Swaminathan. Determining fault tolerance of xor-based erasure codes efficiently. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 206–215, Washington, DC, USA, 2007. IEEE Computer Society.

[43] Q. Xin, E. Miller, and T. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC 2004)*, pages 172–181, June 2004.

[44] Z. Zhang, C. Wang, S. Vazhkudai, X. Ma, G. Pike, J. Cobb, and F. Mueller. Optimizing center performance through coordinated data staging, scheduling and recovery. In *Supercomputing*, Nov. 2007.