On-the-fly Recovery of Job Input Data in Supercomputers

Chao Wang[1], Zhe Zhang[1], Sudharshan S. Vazhkudai[2], Xiaosong Ma[12], Frank Mueller[1]
[1] Department of Computer Science, North Carolina State University Raleigh, NC
[2] Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN
{wchao,zzhang3}@ncsu.edu, vazhkudaiss@ornl.gov, {ma,mueller}@cs.ncsu.edu

Abstract

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 access to the same file remains unaffected. We have implemented 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. Combining parametric and simulation studies, our experiments demonstrate a significant improvement in HPC center serviceability and user job turnaround time.

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 [1]), 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.

In this paper, we propose *on-the-fly data reconstruction* during job execution. We contribute an application-transparent extension to the widely used Lustre parallel file system [2], 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. In a simulation study, using our experimental results as parametric input for recovery overhead and HPC center job traces as workloads, we demonstrate a reduction of over an order of magnitude in the mean wait time of jobs affected by I/O errors.

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.

2. Related work

PFSs like Lustre have active Object Storage Servers (OSSs), each managing many Object Storage Targets (OSTs) and doubling as a hot standby failover node for other OSSs. This obviates the need for redundant OSS nodes. However, upon an OSS failure, the standby OSS node's load practically doubles. Besides cost and performance issues, software compatibility problems prevent the use of storage node failovers, which is the case with Jaguar, the 23,412-core Cray supercomputer at ORNL (No. 7 in Top500). To the best of our knowledge, due to these factors, storage node failover is not widely adopted by supercomputers or clusters. Our approach provides an inexpensive, software-based alternative that protects PFSs against storage node failures by utilizing the natural redundancy in job input data.

While RAID [7] protects against disk failures, it cannot protect against I/O node failures. RAID can also be crippled by multiple disk faults within a group, latent sector errors and controller failure [1, 4, 9]. With increased disk capacity, it is projected that the reconstruction time (already at dozens of hours) will increase by 10% a year [11]. This suggests that a second (non-recoverable) failure is more likely during long reconstructions [11]. Our approach recovers from I/O node failures and could even hide performance degradation due to RAID reconstruction.

I/O shepherding [4] introduces a reliability infrastructure for file systems by executing I/O requests using user-specified failure tolerance 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 failures recovery.

Replication is a commonly used technique for persistent data availability [3, 5, 15]. Supercomputers prefer a high-performance scratch PFS for aggregate I/O bandwidth, which is expensive and, therefore, precious. Replicas consume these precious storage resources as they persist even after job completion. Our recent work assessed the viability of temporally constrained replication [14], but it still comes at the expense of PFS implementation complexity and requires additional scratch space. Our other earlier work recovered lost stagein data *offline* (after job submission but before its dispatch) [16]. The *online recovery* described in this paper complements the latter approach and provides an alternative to replication if its implementation complexity is considered to be too high or when scratch space is scarce.

3. 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.

Architectural Design: To provide fault tolerance to

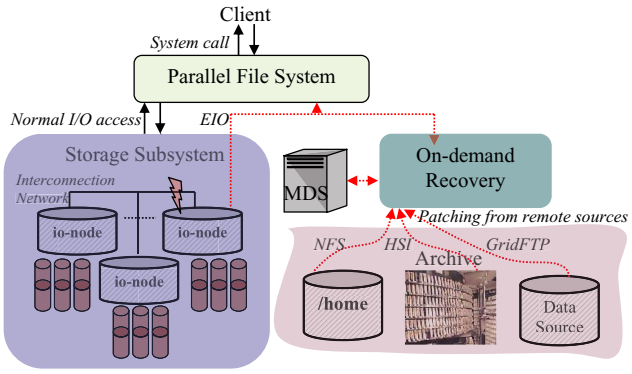


Fig. 1. Architecture of on-the-fly recovery

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 1 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 Metadata Directory Service (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.

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 uniform resource index (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 [16] 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. User credentials, such as GSI (Grid Security Infrastructure) 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.

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 [13]. 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 [12] 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. Although this is beyond the scope of this paper, we have built the basis for a dynamic cost-benefit analysis. Our experiments analyze results and discuss their affect on job turnaround time in light of on-the-fly recovery.

4. Implementation

In this section, we illustrate how on-the-fly recovery has been implemented in Lustre, a modern parallel file system prevalent in supercomputer centers. A Lustre FS comprises of the following three key components: Client, MDS (Meta-Data Server) and OSS (Object Storage Server). Each OSS can be configured to host several OSTs (Object Storage Target) that manage the storage devices (e.g., RAID storage arrays). 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 2 depicts the recovery scenario. Events annotated by numbers happen consecutively in the indicated order resulting in four distinct phases.

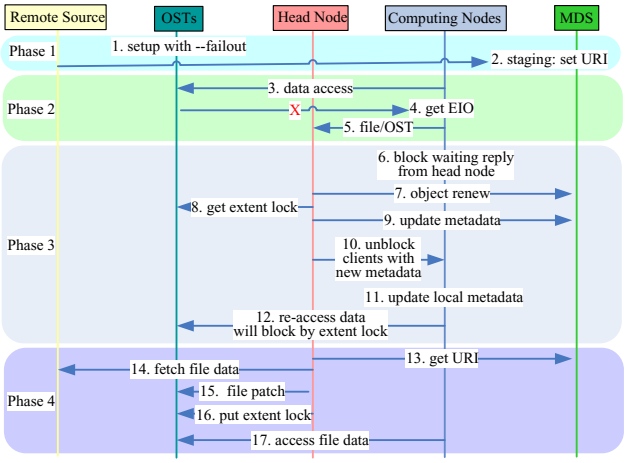


Fig. 2. Steps for on-the-fly recovery

Phase 1: FS 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 2). 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 [16]).

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 [16].

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. 3. 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

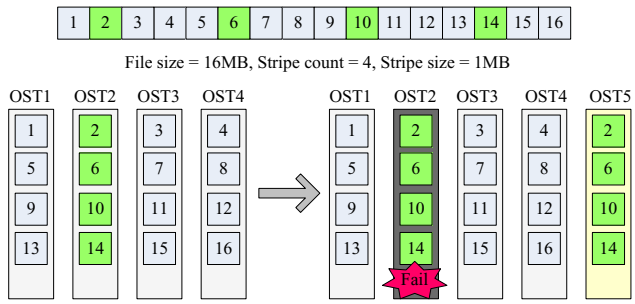


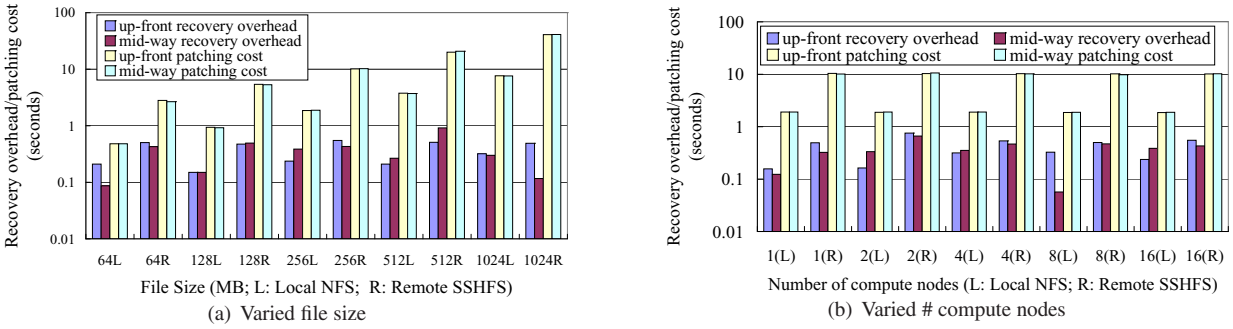
Fig. 3. File reconstruction

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 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.

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



(a) Varied file size (b) Varied # compute nodes

Fig. 4. Matrix multiplication recovery overhead

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).

5. Experimental framework

Our testbed comprised a 17-node Linux cluster at NCSU. Each node was equipped with four 1.76 GHz processing cores (2-way SMPs with dual-core AMD Opteron 265 processors) with 2 GB of memory and connected to a Gigabit Ethernet switch. 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.

6. Experimental results

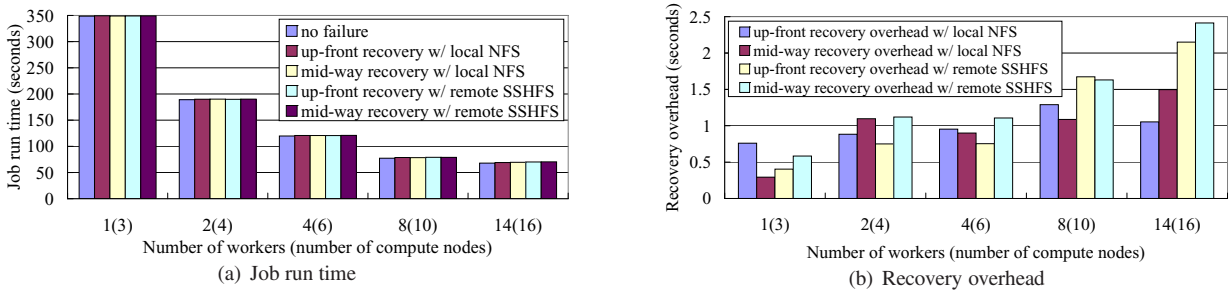
We assessed overhead and patching cost of on-the-fly recovery using an MPI benchmark and an MPI application.

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 4(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 4(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 4(a) and 4(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 4(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.

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



(a) Job run time

(b) Recovery overhead

Fig. 5. mpiBLAST performance

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 5(a) shows the job run time. Figure 5(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 5(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 5(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.

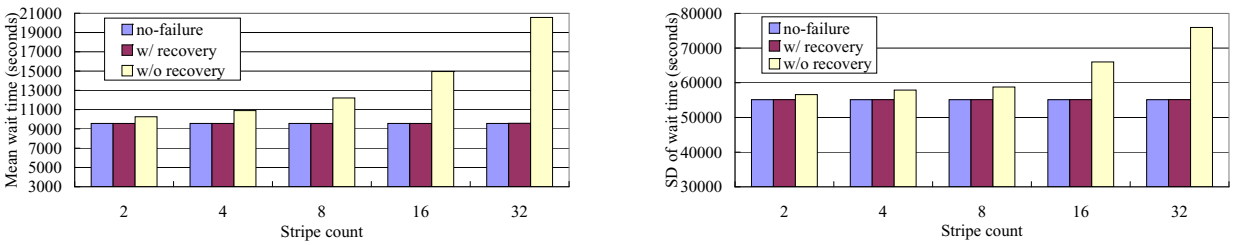
7. Simulation Results

We used the benchmarking results from the previous section in a simulation study considering job traces along with failure traces, both collected by large supercomputer centers. This allows us to study the impact of our approach on overall center performance in terms of the average value and the variance of job wait times.

Setup: We simulate 512 dual-CPU compute nodes (without failures) since we focus on I/O-node and storage failures here. In addition, 72 OSSs serve as I/O nodes, each with two OSTs connected to 8 disk drives (per OST). We use a job trace from LANL system 20 [6], which contains 489,376 job submission and completion records over a period of 1,073 days. Based on the job trace, we generated a set of job submission events, each containing submission time, runtime and number of CPUs per job. For parallel job scheduling, our simulator adopts the FIFO algorithm with backfilling, a popular choice among supercomputing centers.

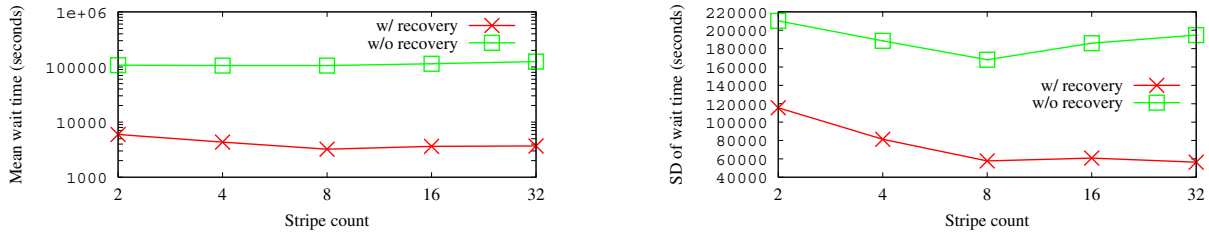
Another trace from LANL for system 20, the node failure trace, contains 2,049 failure records over a period of 1,349 days, each of which indicates the index of the failed node, failure time and duration. In most cluster systems, I/O nodes tend to share the same configuration as compute nodes. Therefore, we extrapolate the failure statistics observed from this trace to the additional I/O nodes. More specifically, system 20's node failure trace is used to calculate the average node failure rate and repair times. We use those statistics to generate a set of failure events for each I/O node. Due to a lack of disk failure data, we derive a common annualized failure rate (AFR) for a storage drive from related work [10, 8]. We randomly choose the failure cases from the node failure trace according to the AFR and apply them to our simulation disks.

As the job trace is devoid of staged data information (*e.g.*, file size, stripe size, stripe count) for each job, we have obtained a snapshot of the Lustre scratch space from ORNL's Jaguar Leadership Computing Facility supercomputer. This staged data trace contains details of every file staged in the scratch PFS. We have calculated the distributions of file sizes, stripe sizes and stripe counts. If I/O operations of jobs overlap with the failure of I/O nodes or drives, the corresponding jobs will experience an I/O error, which triggers recovery. Since the job trace from LANL lacks sufficient information regarding jobs' I/O operation, we assume the worst-case scenario: running jobs are performing I/O operations each time a failure occurs. Whenever a job encounters an I/O failure in our simulation, we charge the patching cost obtained from the previous experi-



(a) Mean wait time (b) Standard deviation (SD) of wait time

Fig. 6. Simulation results of all jobs. Zero-wait jobs are omitted.



(a) Mean wait time (b) Standard deviation (SD) for wait time

Fig. 7. Simulation results of affected jobs

ments as recovery time for data reconstruction from /home.

Results: Our simulations compare system performance with and without recovery for different stripe counts. We use mean and standard deviation (SD) of job wait times to evaluate system performance. Without recovery, if a running job accesses an input file residing on a failed OST, the job exits upon I/O error and is requeued at the queue's tail. Similarly, disk or I/O node failures typically result in job exit before being requeued. With our recovery, jobs will coordinate with the head node to patch missing data and continue to run despite both failure cases.

Figures 6(a) and 6(b) depict the mean and standard deviation (SD) of the wait time for all jobs. Job wait times follow a bimodal distribution with many short and few very long jobs. To address this, we filtered results removing jobs that have a zero wait time under all test configurations. The higher the stripe count, the more OSTs the files are associated with. This means an OST failure will affect more jobs. In fact, the percentage of the affected jobs over all jobs increases from 0.14% to 2.09% when the stripe count increases from 2 to 32. This explains the rise of the curve without recovery with increasing stripe counts. With our recovery mechanism, in contrast, the mean and SD of wait times remain constant as stripe counts increase, indicating a scalable solution for potentially very large Lustre server groups. Furthermore, the recovery mechanism results in the same mean and SD of wait times as the ideal case (without any failure) for all stripe counts.

Figures 7(a) and 7(b) show the mean and SD of wait times for those jobs affected by failures. Since these jobs have non-zero wait times without our recovery, no job filtering is applied. Without recovery, each failed job will be requeued. On-the-fly recovery can handle both failure cases, up-front and mid-way, as mentioned previously. Failure-affected jobs result in slightly longer run times but finish without requeuing. For these affected jobs, gains due to on-the-fly recovery are significant. The mean wait times are reduced by more than an order of magnitude from over 100k seconds to thousands of seconds and show a falling trend (less noticeable due to the log-scale y-axis). In contrast, simulation results with replication resulted in a slight increase in mean wait time and SD for similar settings [14], which underlines the potential of on-the-fly recovery with replication.

Our experiments indicate a system CPU utilization of 70.50% and 70.54% without and with the recovery, respectively. This 0.04% increase is due to the actual recovery of failure-affected jobs. Such an insignificant change can easily be amortized by contemporary HPC systems. This is further reinforced by the observation that under recovery, the same mean and SD of wait times are observed (indicated by the ideal values for all stripe counts).

8. 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. Based on recovery measurements from a real super-

computer, simulation extrapolations of job traces show that the recovery mechanism reduces the mean wait times of jobs from over 100k seconds to thousands of seconds. Thus, 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.

Acknowledgment

This work is supported in part by a DOE ECPI Award (DE-FG02-05ER25685), an NSF HECURA Award (CCF-0621470), a DOE contract with UT-Battelle, LLC (DE-AC05-00OR2275), a DOE grant (DE-FG02-05ER25664) and Xiaosong Ma's joint appointment between NCSU and ORNL.

References

[1] L. Bairavasundaram, G. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *SIGMETRICS*, 2007.

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

[3] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *SOSP*, 2003.

[4] H. Gunawi, V. Prabhakaran, S. Krishnan, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Improving file system reliability with i/o shepherding. In *SOSP*, 2007.

[5] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[6] Los Alamos National Laboratory. Operational data to support and enable computer science research. http://institutes.lanl.gov/data/fdata/, 2006.

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

[8] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST*, 2007.

[9] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. G. abd Andrea C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *SOSP*, 2005.

[10] B. Schroeder and G. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean too you? In *FAST*, 2007.

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

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

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

[14] C. Wang, Z. Zhang, X. Ma, S. Vazhkudai, and F. Mueller. Improving the availability of supercomputer job input data using temporal replication, submitted for publication.

[15] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[16] 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 *SC*, 2007.