Improving the Availability of Supercomputer Job Input Data Using Temporal Replication

Chao Wang* Zhe Zhang* Xiaosong Ma*† Sudharshan S. Vazhkudai† Frank Mueller*

Abstract

Supercomputers are stepping into the Peta-scale and Exa-scale era, wherein handling hundreds of concurrent system failures is an urgent challenge. In particular, storage system failures have been identified as a major source of service interruptions in supercomputers. RAID solutions alone cannot provide sufficient storage protection as (1) average disk recovery time is projected to grow, making RAID groups increasingly vulnerable to additional failures during data reconstruction, and (2) disk-level data protection cannot mask higher-level faults, such as software/hardware failures of entire I/O nodes. This paper presents 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 the parallel file system with the batch job scheduler. We have implemented the temporal replication scheme in the popular Lustre parallel file system and evaluated it with both real-cluster experiments and trace-driven simulations. Our results show that temporal replication allows for fast online data reconstruction, with a reasonably low overall space and I/O bandwidth overhead.

1 Introduction

Coping with failures is a key issue to address as we scale to Peta- and Exa-flop supercomputers. The reliability and usability of these machines rely primarily on the storage systems providing the scratch space. Almost all jobs need to read input data and write output/checkpoint data to the secondary storage, which is usually supported through a high-performance parallel file system. Jobs are interrupted or rerun if input/output data is unavailable or lost.

Storage systems 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 [12]. 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 [20].

Currently, the majority of disk failures are masked by hardware solutions such as RAID [17]. 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 [20].

According to recent studies [**?**], 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 [7], 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 purchase more FLOPS.

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 parallel 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* [13, 16]. 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 the high cost of 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

* Dept. of Computer Science, North Carolina State University {wchao,zzhang3}@ncsu.edu, {ma,mueller}@cs.ncsu.edu

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

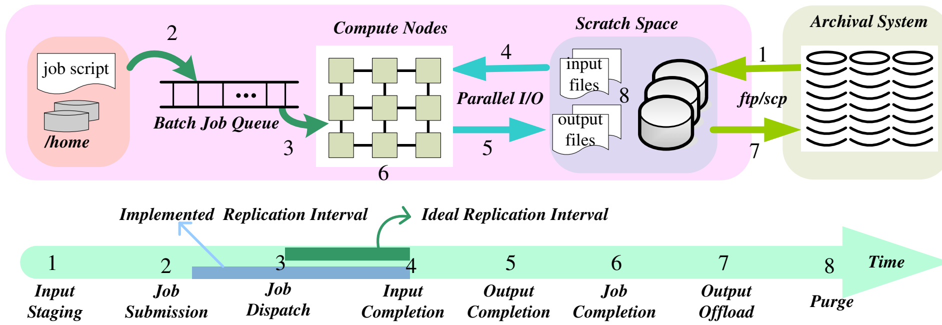


Fig. 1: Event timeline with ideal and implemented replication intervals

selectively provide additional protection only for the duration when the job data is expected to be accessed.

Contributions: In this paper, we propose *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 [30]. We have also implemented a replication-triggering algorithm that coordinates with the job scheduler to delay the replica 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. To evaluate its space overhead, we performed a trace-driven simulation based on three years' worth of detailed job logs obtained from the ORNL Jaguar system (No. 5 in Top500 supercomputers). Our experiments indicate that (1) replication and data recovery can be performed quite efficiently and (2) less than 1% of the total disk scratch space is required to create one extra copy for the input data of active or about-to-be-dispatched jobs. 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.

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 year 2005, to 2.86 in 2006, and 3.84 in 2007.

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 1 summarizes such information for top five supercomputers, on the Top500 list [24]. 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 life span 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 31.8GB, which leads to a bandwidth consumption of less than 0.1% of Jaguar's total capacity of 284GB/s. For this reason, in the following discussions we primarily focus on the space overhead.

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.

System	# Cores	Aggregate 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
Blue Gene/P (Argonne)	163840	80	1126	7.1%	3
Ranger (TACC)	62976	123	1802	6.8%	4
Jaguar Cray XT3/4(ORNL	23412	46.8	600	7.8%	5

Table 1: Configurations of several leading supercomputers as of June 2008

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 1 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 2 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 1: 31.8GB for running jobs and 49.5GB for queued jobs.

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.

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 till 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 [30].

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:

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

where $|Q_i|$ is the number of nodes requested by the ith ranked job in the queue, R is the maximum single-job node footprint (the number of nodes used by the largest running job), S is the total number of nodes in the system, and the replication 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. Our experiments in Section 4.5 provide an empirical study of the choice of α and T.

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 [30]. 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 consider the reduced space requirements for the more common case outweighs this risk.

3 Implementation Issues

A Lustre [7] file system comprises of three key components: clients, a MetaData Server (MDS), and Object Storage Servers (OSS). Each OSS can host several Object Storage Targets (OST) that manage the storage devices. 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.

3.1 Replica Management Services

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,

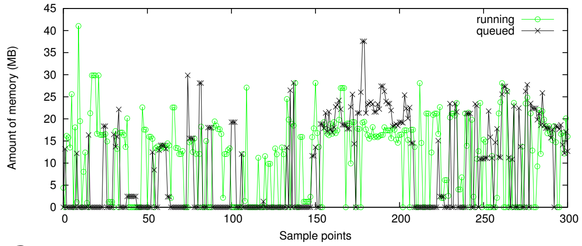


Fig. 2: Per-node memory usage from 300 uniformly sampled time points over a 30-day period based on job logs from the ORNL Jaguar system. For each time point, the total memory usage is calculated as the sum of peak memory used per job, i.e., aggregated across all jobs in question.

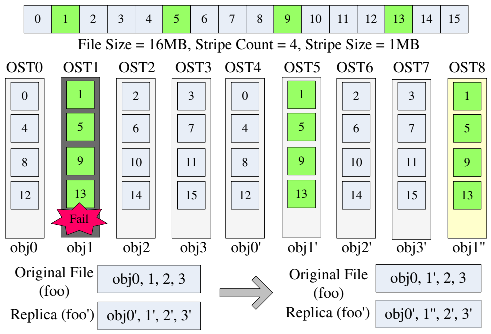


Fig. 3: Objects of an original job input file and its replica. A failure occurred to OST1, which caused accesses to the affected object to be redirected to their replicas on OST5, with replica regeneration on OST8.

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.

3.1.1 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 3, the objects of the original file and the replica form pairs (objects $(0, 0')$, $(1, 1')$, etc.). To locate the replica and its objects, we utilize Lustre's extended attribute mechanism. For a given input file, we add replica details using these attributes. This way, the replica is associated with the original file for its lifetime.

3.1.2 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 [30]. The same mechanism has been extended for transparent storage failure detection and access redirection during a job run. To do so, the POSIX file I/O API is intercepted by our Lustre patched VFS system calls. *E.g.*, the read method of a Lustre client will issue the $ll\_file\_read()$ function. Both I/O node failures and disk failures will result in an I/O error immediately within $ll\_file\_read()$. 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.

3.1.3 Object Failover and Replica Regeneration

Upon an I/O node failure, either detected by the head node as part of 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. The replica management service modules on the head node will query Lustre using several new commands that we have developed, 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).

3.1.4 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. Also, all the concerned compute nodes must receive the same object information to update their local data structure. We have implemented a centralized coordinator inside the replica manager on the head node to handle the requests in a non-redundant fashion.

3.2 Coordination with Job Scheduler

As we discussed in Sections 1 and 2, our temporal replication mechanism is required to be coordinated with the batch job scheduler to achieve selective protection for "active" data. In our target framework, batch jobs are submitted to a *submission manager*, which parses the scripts, recognizes and records input data sets for each job, and creates corresponding replication operations at the appropriate time.

To this end, we leverage our previous work [30] that automatically separates out data staging and compute jobs from a batch script and schedules them by submitting these jobs to separate queues ("dataxfer" and "batch") for better control. This enables us to coordinate data staging alongside computation by setting up dependencies such that the compute job only commences after the data staging finishes. The data operation itselfis specified in the PBS job script as follows using a special "STAGEIN" directive:

```
  #STAGEIN hsi -q -A keytab -k
my_keytab_file -l user
  ''get /scratch/user/destination_file :
input_file''
```

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

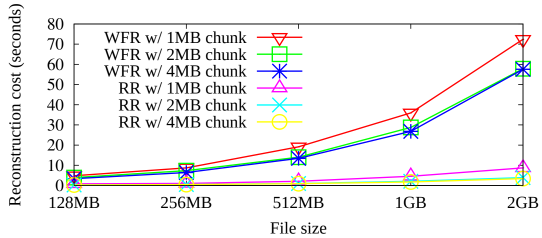


Fig. 4: Offline replica reconstruction cost with varied file size

We extend this work by setting up a separate queue, "ReplicaQueue", that accepts replication jobs. We have also implemented a *replication daemon* that determines "what and when to replicate". The replication daemon creates a new replication job in the ReplicaQueue so that it completes in time for the job to have another copy of the data when it is ready to run. The daemon periodically monitors the batch queue status using the *qstat* tool and executes the delayed replica creation algorithm described in Section 2.2. These strategies enable the coordination between the job scheduler and the storage system, which allows data replication only for the desired window during the corresponding job's life cycle on a supercomputer.

4 Experimental Results

To evaluate the temporal replication scheme, we (1) performed real-cluster experiments and (2) conducted a trace-driven simulation study. The former assesses our implementation of temporal replication in the Lustre file system, in terms of the online data recovery efficiency. Two representative parallel codes are used to measure the visible overhead of failure detection and data reconstruction. The latter assesses the long-term, center-wide scratch space consumption of temporal replication for job input data based on three years of job logs collected from the ORNL Jaguar system.

4.1 Experimental Framework

Our testbed comprised a 17-node Linux cluster at NCSU. The nodes were 2-way SMPs each with four AMD Opteron 1.76 GHz cores and 2 GBs of memory, connected by a Gigabit Ethernet switch. The OS used was Fedora Core 5 Linux x86_64, with Lustre version 1.6.3. The cluster nodes were setup as I/O servers, or compute nodes (Lustre clients), or both, as discussed later.

4.2 Failure Detection and Offline Recovery

As mentioned in Section 3.1.2, before a job begins to run, we periodically check for failures on OSTs that carry its input data. As we configure all OSTs in the "fail-out" mode, OST failure can be recognized without any timeout. 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, the following 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 4 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 4 also shows that the cost of *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.

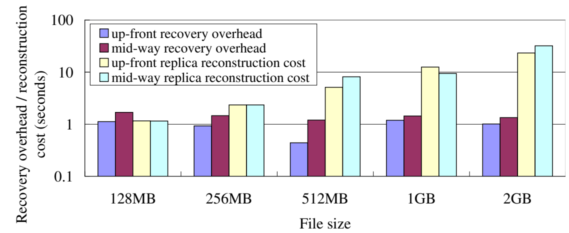


Fig. 5: MM recovery overhead vs. replica reconstruction cost

4.3 Online Recovery

4.3.1 Application 1: Matrix Multiplication

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. Since input operations are concentrated at the beginning of the run, and the code is computation-intensive, we focus on the visible recovery overhead with 1 OST failure.

Figure 5 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. The input file *stripe count* was 4, and the *stripe size* was 1MB. 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 the other case, "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 5 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 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.

4.3.2 Application 2: mpiBLAST

To evaluate the data recovery overhead using temporal replication with a read-intensive application, we tested with

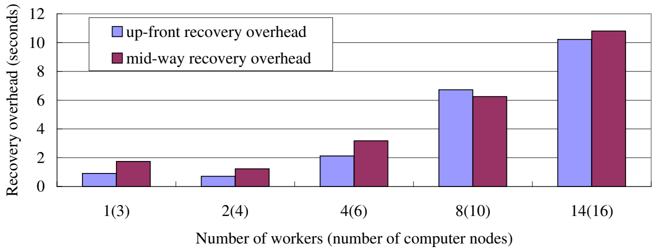


Fig. 6: **Recovery overhead of mpiBLAST**

mpiBLAST [9], a well-known parallel sequence database search tool. mpiBLAST splits a database into fragments and performs 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 6. Since the version of mpiBLAST we used 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. Each worker process read several database fragments.

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 6, the recovery overhead grew with the number of workers.For this application, recovery of the failed files was not conducted in parallel. 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 much more scalable recovery cost with regard to the number of MPI processes.

4.4 Trace and Simulation Overview

The real-cluster results presented in the previous sections demonstrated the efficient online data recovery due to temporal replication by examining individual jobs. To assess the overall space overhead caused by temporal replication, we need to examine the impact of this policy on all jobs for an extended period, collectively. To this end, we conducted extensive trace-driven simulations. Our simulation was performed using the operational data from the ORNL Jaguar supercomputer (currently No. 5 in the top500 list) with job logs collected from Apr. 2005 to Nov. 2007. Each job entry contains timing information (such as submission, dispatch, and completion times) and resource usage details (such as the number of cores requested and the peak memory usage per core).

One limitation of the Jaguar job log is that it is devoid of information on job input data size. This metric is needed to determine the amount of storage space required for replicas. However, as discussed in Section 2.1 and supported by the memory usage pattern of jobs on Jaguar shown in Figure 2, we can safely estimate this information based on each job's peak aggregate memory usage.

4.5 Replication Simulation Results

In our experiments, we replayed the job traces and simulated a job queue according to the submission and dispatch times of jobs. Several replication strategies, described in Section 2.2, were used to evaluate our dynamic replica creation algorithm to study the balance between enhanced data availability and increased space usage. The first one was FCFS, where the replication window is calculated based on the arrival time of jobs. The next two strategies used the shortcut approach, using a T value of 0.01 (FCFS with small threshold) and 0.1 (FCFS with large threshold), respectively. Finally, as a reference, we used an offline algorithm (dispatch-aware) that was aware of the actual dispatching order of jobs a priori and calculate the replication window accordingly. This offline algorithm indicates the best decision that can be made with the replication window approach.

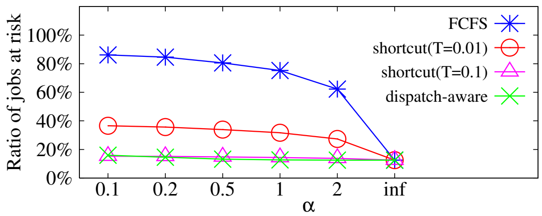
Figure 7 compares the above algorithms using different α levels. The $alpha$ value "inf" corresponds to an infinite replication window size, where all jobs' input datasets are replicated upon submission regardless of the replication strategy (hence the convergence).

Figure 7(a) shows the percentage of jobs that are "at risk". A job was considered at risk if its replica was not ready when the job was dispatched. The replication time for each input dataset was calculated using the cost of a file copy, benchmarked on Jaguar via the command `cp`. Due to the small default Lustre stripe width, a large number of available OSSs, and the low average number of concurrent replication operations estimated in our simulation, we did not consider the impact of concurrent replications on the copy bandwidth.

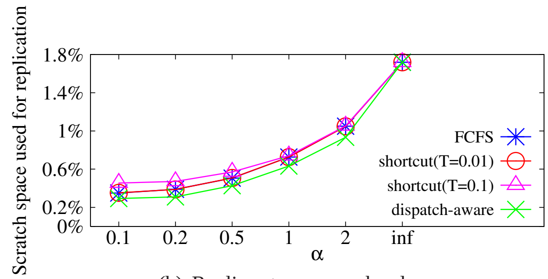
With the basic FCFS replication strategy, a large fraction of jobs are at risk with small α levels, and even with a relatively large α level of 2, more than 60% of jobs will not have their replicas created early enough. The offline algorithm (dispatch-aware), in contrast, is highly insensitive to α. This reveals that to have an appropriate level of data redundancy, only a small portion of jobs in the queue needs to be selected for replication if we know the exact order of dispatched jobs. The dispatch-aware curve is almost flat due to the existence of jobs that are scheduled immediately upon submission, which leaves no time for data replication. Fortunately, these jobs tend to be debug or testing jobs that request a small number of nodes and run time, making them less vulnerable to failures and requesting a small resubmission cost.

With the shortcut enhancement to FCFS, we see a very selective threshold of 0.01 will dramatically reduce the jobs at risk. A threshold of 0.1 will generate a fault tolerance level virtually identical to the offline dispatch-aware algorithm.

In Figure 7(b), we compare the space overhead of the above strategies by calculating the ratio of total scratch space used for storing job input file replicas. This ratio was averaged over 300 snapshots, regularly sampled during the 32-months-long period. For each snapshot, the total replica size was estimated as the sum of the peak aggregate memory usage of all jobs that have input data replicas created, including both running and waiting jobs. The results indicate that, overall, creating one extra copy of active jobs' input data consumes a very limited fraction of the entire scratch space on Jaguar.



(a) Jobs at risk



(b) Replica storage overhead

Fig. 7: Simulation results using a 3-year ORNL Jaguar log with replication covering jobs up to αS nodes; Infinity (inf) denotes replication for all Jobs.

Even with the most aggressive replication setting (an infinite α level), the replicas will, on average, occupy less than 1.8% of the scratch space. The space overhead significantly reduces as the α value is lowered. The differences between different strategies are quite small since they mainly differ in the handling of small jobs. The dispatch-aware algorithm has an edge due to its optimal job selection.

In summary, considering both aspects shown in Figure 7, we regard that FCFS with small threshold as an effective replication strategy for our traces. A threshold of 0.1 will generate close-to-ideal data redundancy, which, in turn, allows a small replication window of α levels such as 0.2 and 0.5. Such small windows will incur very low space overhead, lower than 0.5% of the scratch space.

Finally, Figure 8 demonstrates the behavior of the replication strategies under different job workload levels, by plotting the percentage of jobs at risk for the same month (August) in three consecutive years. As mentioned earlier, most supercomputers tend to get crowded once into production. For our trace, the average job queue length increases from 1.5 in 08/2005, to 27.5 in 08/2006, and to 45.5 in 08/2007. Under a light load, most jobs can be scheduled quickly, without a sufficient enough waiting period required for just-in-time replication regardless of strategies. As the system gets busier, different strategies begin to diverge. While the naive FCFS strategy produces a high at-risk rate, short cut with T=0.1 stays close to the offline dispatch-aware algorithm and is able to replicate most datasets in time. This result further justifies the advantage of temporal replication when the system is busier. A busy system implies longer wait times and, consequently, longer turnaround time for jobs that are required to be resubmitted due to input data unavailability as they get back at the end of the queue. However, the longer queue wait time provides enough room for just-in-time replication.

5 Related Work

RAID recovery: Disk failures can often be masked by standard RAID techniques [17]. However, RAID is geared toward whole disk failures and does not address sector-level faults [2, 11, 19]. 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 [29, 22]. 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 [20]. Our approach complements RAID systems by providing fast recovery, protecting against non-disk and multiple disk failures.

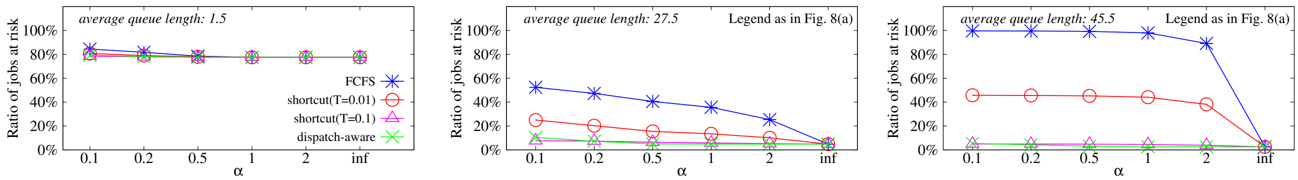
Recent work on popularity-based RAID reconstruction [23] rebuilds more frequently accessed data first, thereby reducing reconstruction time and user-perceived penalties. However, supercomputer storage systems host transient job data, where "unaccessed" job input files are often more important than "accessed" ones. In addition, such optimizations cannot cope with failures beyond RAID's protection at the hardware level.

Replication: Data replication creates and stores redundant copies (*replicas*) of datasets. Replication has stronger fault tolerance than RAID because replicas of a dataset reside on independent components in the system and have a smaller chance of simultaneous failure. Various replication techniques have been studied [4, 8, 21, 27] in many distributed file systems [1, 5, 10, 14].

Most existing replication techniques treat all datasets with equal importance and each dataset with static, time-invariant importance when making replication decisions. This indiscriminate replication increases storage space consumption manifold, which is a significant burden for heavily loaded storage systems. An intuitive improvement would be to treat datasets with different priorities. To this end, BAD-FS [3] performs selective replication according to a cost-benefit analysis based on the replication costs and the system failure rate. Similar to BAD-FS, our approach also makes on-demand replication decisions. However, our temporal replication scheme is more "access-aware" rather than "cost-aware". While BAD-FS still creates static replicas, our approach utilizes explicit information from the job scheduler to closely synchronize and limit replication to jobs in execution or soon to be executed.

Erasure coding: Another widely investigated technique is erasure coding [6, 18, 28]. 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 [26], 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 approach is more suitable for supercomputing environments, which is verified by our experimental study.



(a) August of 2005 (b) August of 2006 (c) August of 2007

Fig. 8: The ratios of jobs at risk in the month of August in 2005, 2006 and 2007.

Remote reconstruction: Some of our previous studies [15, 25, 30] investigated approaches for reconstructing missing pieces of datasets (either due to cache miss or failures) from data sources where the job input data was originally staged from. By modifying the parallel file system to record the data source location as a file metadata item, such remote data reconstruction can be performed on demand and in a transparent fashion. We have shown in [30] 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.

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 may be hard to hide them from applications during a job's execution. Temporal replication, on the other hand, trades space (which, we argue, 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. Our optimizations presented in this paper aim at further controlling and lowering the space consumption of replicas.

6 Conclusion

In this paper, we have presented a novel temporal replication scheme for supercomputer job data. By creating additional data redundancy for transient job input data and coordinating the job scheduler and the parallel file system, 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. Our simulation study, using a 32-month job trace collected from a top10 supercomputer, demonstrates that by limiting the replication to active or about-to-be-dispatched jobs, we can afford to create one extra copy for their input data using less than 1% of the total disk scratch space.

Acknowledgment

This research is sponsored in part by the National Center for Computational Sciences at Oak Ridge National Laboratory (ORNL), managed by UT-Battelle, LLC for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

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

[2] Lakshmi Bairavasundaram, Garth Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, pages 289 – 300, June 2007.

[3] 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*, March 2004.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] Aaron E. Darling, Lucas Carey, and Wu 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.

[10] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *Proceedings of the 19th Symposium on Operating Systems Principles*, 2003.

[11] H Gunawi, V. Prabhakaran, S. Krishnan, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Improving file system reliability with i/o shepherding. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, October 2007.

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

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

[14] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael 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.

[15] 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.

[16] H. Monti, A.R. 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.

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

[18] 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.

[19] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi abd Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206 – 220, October 2005.

[20] B. Schroeder and G. Gibson. Understanding failure in petascale computers. In *Proceedings of the SciDAC Conference*, 2007.

[21] 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.

[22] Alexander Thomasian, Gang Fu, and Chunqi Han. Performance of two-disk failure-tolerant disk arrays. *IEEE Transactions on Computers*, 56(6):799–814, 2007.

[23] Lei Tian, Dan Feng, Hong Jiang, Ke Zhou, Lingfang Zeng, Jianxi Chen, Zhikun Wang, and Zhenlei Song. Pro: a popularity-based multithreaded reconstruction optimization for raid-structured storage systems. In *FAST'07: Proceedings of the 5th conference on USENIX Conference on File and Storage Technologies*, pages 32–32, Berkeley, CA, USA, 2007. USENIX Association.

[24] Top500 supercomputer sites. http://www.top500.org/, June 2007.

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

[26] 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.

[27] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*, November 2006.

[28] Jay J. Wylie and Ram 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.

[29] 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.

[30] Z. Zhang, C. Wang, S. S. Vazhkudai, X. Ma, G. Pike, J. Cobb, and F. Mueller. Optimizing center performance through coordinated data staging, scheduling and recovery. In *Proceedings of Supercomputing 2007 (SC07): Int'l Conference on High Performance Computing, Networking, Storage and Analysis*, November 2007.