/Scratch as a Cache: Rethinking HPC Center Scratch Storage

Henry M. Monti, Ali R. Butt
Department of Computer Science
Virginia Tech.
Blacksburg, VA 24061 USA
{hmonti, butta}@cs.vt.edu

Sudharshan S. Vazhkudai
Oak Ridge National Laboratory
Oak Ridge, TN 37831 USA
vazhkudaiss@ornl.gov

ABSTRACT

To sustain emerging data-intensive scientific applications, High Performance Computing (HPC) centers invest a notable fraction of their operating budget on a specialized fast storage system, scratch space, which is designed for storing the data of currently running and soon-to-run HPC jobs. Instead, it is often used as a standard file system, wherein users arbitrarily store their data, without any consideration to the center's overall performance. To remedy this, centers periodically scan the scratch in an attempt to purge transient and stale data. This practice of supporting a cache workload using a file system and disjoint tools for staging and purging results in suboptimal use of the scratch space.

In this paper, we address the above issues by proposing a new perspective, where the HPC scratch space is treated as a cache, and data population, retention, and eviction tools are integrated with scratch management. In our approach, data is moved to the scratch space only when it needed, and unneeded data is removed as soon as possible. We also design a new job-workflow-aware caching policy that leverages user-supplied hints for managing the cache. Our evaluation using three-year job logs from the Jaguar supercomputer, shows that compared to the widely-used purge approach, workflow-aware caching optimizes scratch utilization by reducing the average amount of data read by 9.3%, and by reducing job scheduling delays associated with data staging, on average, by 282.0%.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]; C.4 [**PERFORMANCE OF SYSTEMS**]: Reliability, availability, and serviceability; D.4.2 [**Storage Management**]: Allocation/deallocation strategies

General Terms

Design, Experimentation, Performance

Keywords

HPC scratch management, just-in-time staging, offloading

ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.

1. INTRODUCTION

The scratch file system in an High-Performance Computing (HPC) center provides fast temporary local storage space for jobs, often consuming a notable fraction of the center's operations budget. Scratch storage is intended for very large—typically on the order of terabytes—input, output, and intermediate data of currently running and soon-to-run user jobs. This storage is usually served via a parallel file system (PFS) that supports very high aggregate I/O throughput, e.g., Lustre [10], PVFS [9], and GPFS [23]. Consequently, to ensure efficient data I/O and to support improved job turnaround times, supercomputing application programmers are encouraged to utilize the scratch space.

The scratch space, however, impacts the HPC center's serviceability and necessitates proper provisioning to accommodate the storage demands of all incoming jobs. Unlike the user "home" file system that is meant for application development, the scratch is seldom regulated with quotas to avoid introducing any ceilings on the applications' data sizes. The input datasets are brought in from remote data sources, and the result files are offloaded to end-user and other off-center destinations. Consequently, the scratch storage is designed to be a staging ground for transient datasets that are usually not held beyond the lifetime of a job run. However, due to the lack of sophisticated "end-user data services"–timely staging of input data and offloading of result output data– HPC centers often resort to "purge" mechanisms that sweep the scratch space to remove files found to be no longer in use, based on not having been accessed in a pre-selected time threshold called purge window that commonly ranges from a few days to a week. The purge window depends on the load, total scratch size, and required level of serviceability.

The scratch space is not intended to be used as a generic file system for persistent user files storage. Instead, it is a special-purpose storage for needed ("hot") data of running and waiting jobs. Nonetheless, in practice, the scratch space is utilized as a traditional file system, with the purge policy added as an afterthought to delete unneeded ("cold") data of finished jobs and to cap the scratch utilization within limits. Such an approach has several disadvantages. First, it wastes scratch space by allowing users to stage input data much earlier than job commencement and offload results much later than job completion. This leads to sub-optimal use of scratch space, which should be used for new incoming jobs. By extension, this impacts the HPC center's serviceability. Second, it renders the input and output data vulnerable to scratch storage system failure during the extra wait time, which can increase job turnaround time.

The lack of elegant scratch space management is having a profound impact on HPC centers. Users arbitrarily stage and offload data as and when they deem fit, without any consideration to the center performance. Few solutions that are available in this landscape (e.g., the purge mechanism) are disjoint with user job workflow, and thus are not efficient. Further, users can easily trick the purging system into not deleting their datasets by periodically "touching" the datasets, and essentially rendering the purge ineffective. There do exist workflow-aware data transfer approaches [18, 19], however, these support standalone operation only, and lack tight integration with scratch management. Thus, there is an urgent need for a coherent scratch space management solution. Such an approach can be very timely when it comes to HPC acquisition proposals. Multi-million dollar HPC acquisition proposals are won based on the FLOPS provided. Every dollar spent on provisioning the scratch space is a dollar taken away from buying FLOPS. Efficient management can transform the productivity of even an under-provisioned scratch storage system.

To address these issues, we present a fresh perspective to scratch storage management by fundamentally rethinking the manner in which scratch space is employed. Our approach is to re-design the scratch system as a "cache" and build "retention" and "eviction" policies that are tightly integrated from the start, rather than being add-on tools. We have built cache retention and eviction policies using "hints" from the user's job submission script in order to accurately capture the data needs of a job workflow. These hints include information about job input, output, and intermediate files, their usage duration and the dependencies of other pieces of the workflow on these datasets. Such a strategy allows us to couple job scheduling with cache management, thereby bridging the gap between system software tools and scratch storage management. It enables the retention of only the relevant data for the duration it is needed. Redesigning the scratch as a cache captures the current HPC usage pattern more accurately, and better equips the scratch storage system to serve the growing datasets of workloads. This is a fundamental paradigm shift in the way scratch space has been managed in HPC centers, and outweighs providing simple purge tools atop a file system and using that to serve a caching workload.

While staging and offloading have been examined in isolation previously, existing works [19, 18] lack a holistic approach to scratch management, and only address automatic data movement alongside computation on a per job basis. In contrast, we provide center-wide global optimizations by deriving hints from every job workflow.

Our evaluation using simulations driven by logs collected over a period of three years at the Jaguar supercomputer (No. 2 in Top500), shows that compared to the purging approach, workflow-aware caching reduces average scratch utilization per hour by 6.6% compared to LRU based caching, reduces the average amount of data read by 9.3%, and provides, on average, 282.0% reduction in job scheduling delays associated with data staging.

2. RETHINKING "HOT" AND "COLD" CONTENTS

Managing scratch as a cache entails rethinking the traditional classification of cache contents as "hot" and "cold". Typically, the most recently used (MRU) dataset is considered hot and retained in the cache as it is likely to be accessed again. Conversely, the dataset that is least recently used (LRU) is considered cold, and is evicted from the cache. However, this classification does not always apply to scratch space contents. Below, we highlight some common scenarios and discuss hot and cold in the context of "scratch as cache".

An input dataset consumed by a job that has completed. Even though the dataset was recently used, it is unlikely that it will be reused by any other job on the supercomputer. In fact, most HPC jobs consume their input data during the initial phase of the run and do not reuse it again. These most recently used datasets are thus cold and can be evicted.

A dataset that was recently staged into scratch in anticipation of a job run. This is an MRU dataset that is hot and cannot be evicted as the associated job has not even started. However, if the job run is delayed, under traditional scratch operations, the user will need to explicitly touch the dataset periodically to avoid purging. In modern (crowded) HPC centers, long job wait times are the norm.

A dataset that the user simply "touches" to persistently avoid purging and trick the system. These MRU datasets may be cold and can be evicted if no jobs use them as input. Traditional scratch operations cannot identify, and catch this scenario.

Result and intermediate datasets that were recently produced. These MRU datasets can be evicted from the scratch space, as long as they are not input to other co-dependent jobs in the workflow.

An input dataset that is cold due to prolonged job wait and the user has not been renewing it via re-touching. This is an LRU dataset that is hot and should not be evicted. To begin with, this input dataset should only have been brought into scratch storage to coincide job startup.

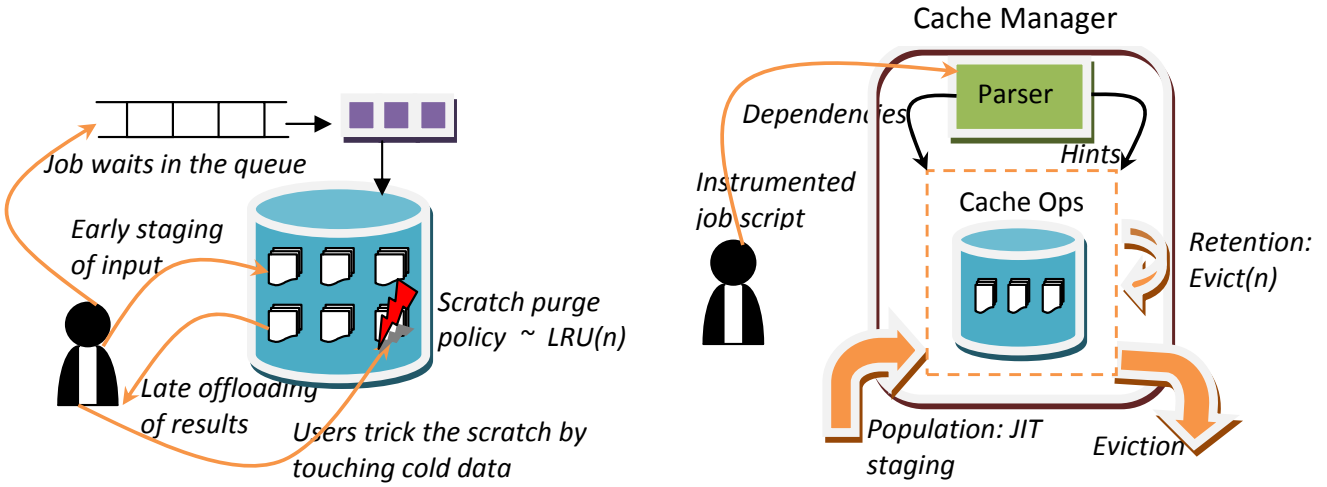
This shows that access recency or frequency are not the only reasons driving scratch management. The aforementioned scenarios require *information from the job workflow to identify truly hot and cold contents*. Note that traditional scratch management cannot even capture these usage scenarios.

2.1 Goals

In light of the above discussion, we foresee several objectives for a system that manages the HPC center scratch storage as an advanced cache. Namely:

- **Optimize scratch space consumption.** From a center standpoint, it is desirable to stage the data of a waiting job as late as possible so that the precious scratch space is available for all of the currently running jobs' I/O (e.g., checkpointing and output). Reducing the waiting jobs' duration of scratch usage will enable the center to service the currently running jobs better.

- **Reduce exposure window.** Another downside of staging the data early is the prolonged exposure of staged data to potential storage system failure. We refer to the time elapsed between when data is staged until the job starts running as exposure window, E_w.



(a) *Traditional approach to scratch usage.*

(b) *Scratch as a Cache.*

Figure 1: (a) Traditional approach to scratch usage is a set of disjoint tools, performing un-coordinated data movement, job submission, and scratch purging. (b) Scratch as a cache operates the scratch using specific cache operations that are driven using hints from the job workflow.

To protect against storage failures, it is desirable to reduce E_w, preferably as close to 0 as possible.

- **Reduce job stall time.** The previous goals support delaying staging of input data to the scratch, however, such delay can result in jobs being stalled and possibly rescheduled—a costly process–, as they wait for the necessary input data to become available. Thus, management of scratch as a cache also requires utilizing techniques such as intelligent prefetching to timely stage job input data.

3. SCRATCH AS A CACHE

The key idea behind our approach is to view the HPC scratch storage as a cache. To this end, we need to support three basic cache operations namely, "populating" the cache, "retaining" appropriate datasets in the cache, and "evicting" datasets from the cache. The premise is that if these operations are integrated with the storage system, and are the basis for its functioning, then scratch space usage can be fundamentally optimized. The problem, as mentioned earlier, with the way scratch storage is currently managed in HPC centers is that the data purge operations are added as an afterthought and are not workflow-aware and lack tight integration with scratch management: Data is staged to/from the scratch in an out-of-band way, and purging is the only means to ensure space availability.

The purge mechanism performs an LRU-like eviction based on a temporal window, deleting datasets that have not been accessed for the last 'n' days [2]. However, this approach is unable to capture many of the discussed scratch usage scenarios. This is because the HPC job workflow and the legacy of the user job submission process imposes new requirements on the scratch, which cannot be satisfied by a purge policy alone. Troublesome scenarios include when users stage in data way in advance, or offload result data much later, as well as user behavior (touching files) to mitigate the effects of un-coordinated job submission process.

Figure 1(a) presents an overview of how scratch space is used currently in HPC centers, using a set of disjoint tools (out-of-band data movement, job submission, and purging) with no coordination between one another. Moreover, the figure depicts a number of individual operations manually performed by the user on the scratch storage. The problem is only compounded with ever increasing users, each performing such disconcerted operations. In crowded HPC centers, where scratch space is precious, streamlining usage by way of treating the scratch as a cache can improve serviceability. On one end, using this approach, even modestly provisioned scratch storage systems can be tuned to perform optimally. At the other end, leadership-class facilities that boasts several hundred terabytes of scratch space can also benefit from sophisticated scratch storage management, as modern petascale applications at such centers consume ever more data.

3.1 Cache Management Overview

Figure 1(b) presents an overview of scratch as a cache. In this approach, direct user managed operations on the scratch storage is avoided and the scratch is strictly managed using cache management tools. User submitted jobs are translated into a series of cache operations, in addition to the computation itself, which are then used to operate the scratch storage. We manage the scratch cache by ensuring that all staging and offloading of job data is performed using cache population and eviction tools. In this model, users do not arbitrarily move data in and out. Job input and output data are not retained beyond the lifetime of the application run, unless otherwise specified. Populating the cache with job input data is accomplished using just-in-time staging tools so that it coincides with job startup. This ensures that the input data is not moved into the scratch space too much in advance, occupying space and increasing the exposure window (E_w). Only data that is needed immediately is retained in the cache. Cache eviction involves offloading result files immediately after the computation has finished.

Thus the output data of a job is not held in the scratch cache beyond the lifetime of the job run. Consequently, the cache is strictly used for hot job data. Cold data, even though only recently produced (output), is moved out of the cache.

Compared to the traditional way of scratch usage (Figure 1(a)), the cache approach significantly reduces the direct user interaction with scratch storage. Each user's job is now streamlined into a well coordinated set of operations that are performed by the center as and when it is optimal to do so rather than disjoint activities. This significantly optimizes scratch space usage and makes it more available for running or soon-to-run jobs. An additional advantage is that we can now perform globally optimal decisions that improve the HPC center performance at large.

The logical extension to the "scratch as cache" paradigm is to view the scratch as one of the levels (Tier 1) within a multi-level storage for the HPC center. The next level, Tier 2, can be more broadly defined as a variety of potential sources and destinations for the job datasets, including center-wide storage [5, 1], archives [11] at the center, user-specified nearby storage [22, 19], or end-user location. Data is moved into and out of Tier 1 from/to Tier 2 storage using cache management tools.

4. WORKFLOW-DRIVEN CACHING

From the above discussion, it is evident that in order to manage and operate the scratch as a cache we need guidelines and "hints" from the user's job workflow. The job workflow can provide details such as input, output, and intermediate files, their sources and destinations, the transfer protocols to be used, and more importantly crucial data dependencies. For instance, the workflow can be used to garner information such as whether the output of one task is input to another. Such a dependency can be used to determine if a given output dataset should be retained in the cache. Workflow-specific hints enable retention of datasets only for the duration they are needed. Current scratch operations are significantly stymied by the lack of such coordination between user workflows and scratch storage management, which results in un-coordinated data movement, wastage of scratch space and, potentially, increased job turnaround and a negative impact on HPC center serviceability. Workflow-driven cache management can remedy such issues and improve serviceability.

4.1 Collecting Information from the Job Script

HPC users normally specify their resource requirements and data movement in a job script and submit it to the job scheduler at the center. The resource manager at the center deciphers these requirements, allocates resources, and executes the data movement and computation commands. Therefore, the job script is a logical place to specify hints that can aid in cache management. If we can instrument the job script with guidelines regarding which input datasets of the job to populate the cache with, which ones to evict, and which ones to retain and for how long, then the cache management infrastructure could use this information to make global decisions across all jobs.

4.1.1 Instrumenting the Job Script

To support cache management, we have instrumented the PBS [20] job scripting system with cache-specific directives. Users can prefix the data movement operations that they already conduct with `#Populate`, `#Evict`, and `#Retain` directives to indicate the input and output files, their sources, destinations and transfer protocols in Tier 2 storage. Below is a sample PBS script with the directives to populate Tier 1 from Tier 2 using the `gridftp` transfer protocol, to evict from Tier 1 to Tier 2 using `scp`, and to retain an output dataset for a certain duration and eventually evict it to Tier 2.

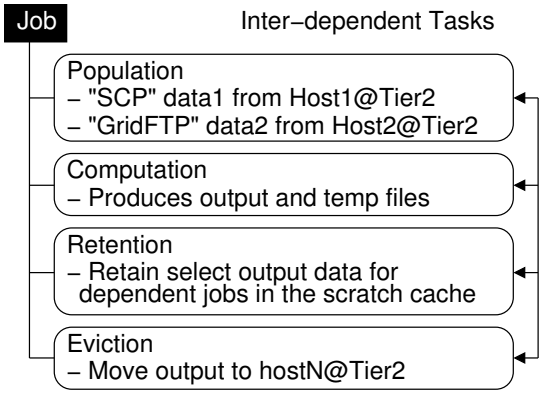


Figure 2: A single user job being parsed into population, computation, retention, and eviction jobs.

```
#PBS -N SampleJob
#Populate gridftp://Tier2/home/user/InputFile1
                -l user file://Tier1/scratch/user
#Populate gridftp://Tier2/home/user/InputFileN
                -l user file://Tier1/scratch/user
mpirun -np 128 /myapp
#Evict file://Tier1/scratch/user/Output1
            scp://Tier2/home/user/Output1 -l user
#Evict file://Tier1/scratch/user/OutputN
            scp://Tier2/home/user/OutputN -l user
#Retain file://Tier1:/scratch/user/Output5
  [-d HOURS] -evict scp://Tier2/home/user/Output5
```

4.2 Cache Operations

Instrumented scripts, such as the one shown above, are input to a parser that is part of the cache management suite, which identifies job files, their locations and longevity in the Tier 1 cache. The parser separates the job script into "cache population jobs", "eviction jobs", "retention jobs" and the computation job with dependencies between them so that the cache population occurs before computation commencement and an eviction is only carried out after job completion (Figure 2). To this end, we leverage work, such as [28], which exploit modern resource manager (e.g., PBS [20], Moab [17]) primitives to setup job dependencies for sequencing multiple jobs together. Our work is significantly different in that it uses these dependencies and instrumentation to fundamentally rethink scratch operations as a cache and not just automate data movement. These cache operation jobs (cache-ops) could even be submitted to a separate "CacheOps" queue instead of the standard batch queue used for computational jobs. The queue could be setup to accept only cache-ops that are size zero jobs that usually involve only data movement and will be run on the center's I/O nodes.

4.2.1 Populating the Cache

The cache management examines the submitted jobs to determine when a particular job's population operation should be initiated. Populating the scratch cache is essentially the staging of data from Tier 2 to Tier 1 storage. However, the staging of input data is not performed immediately after job submission as the job may have to wait in the queue until compute resources become available. To this end, the cache population operations perform "just-in-time" (JIT) staging to bring the data in to coincide job startup. Here, we borrow from similar work on data staging [4, 18] that attempted to minimize the data exposure window of an input dataset on the scratch space. In our context, the exposure window (E_w) can be defined as the time spent by the input dataset waiting for the job to commence and is, $E_w = T_{JobStartup} - T_{StageIn}$, where $T_{JobStartup}$ is the estimated job startup time and $T_{StageIn}$ is the time to stage the input dataset. JIT staging makes use of an estimated job startup time, from a batch queue prediction service (e.g., NWS [27]), as a data staging deadline and a dynamic, decentralized transfer scheme that is able to adapt to changing conditions to deliver data in time. For the purpose of this discussion, we assume that a transfer time is specified by the user as part of the populate directive in the job script, much like the "walltime" specified by users to denote the duration of an application run. In the absence of explicit transfer time specification, the cache management tool can also perform on-the-fly bandwidth measurement to the Tier 2 storage to estimate transfer times.

Thus, the cache population jobs are submitted to the appropriate queue in the resource manager and are launched as late as possible so as to minimize E_w. During this time, the computation job is submitted to the batch queue so it can commence execution by $T_{JobStartup}$, but with a dependency on the input data population job.

4.2.2 Evicting from the Cache

Evicting the output data of a job is essentially offloading it out to Tier 2 storage. The transfer protocol and authentication to be used for this operation is provided as hints in the `#Evict` directive in the job script. The eviction job is configured to begin immediately after the completion of the compute job. In addition to moving the result output data, the user can also identify "temp" files of the application run that are no longer needed. Temporary files from a petascale application run can amount to several terabytes of data themselves. In many cases, these are used for debug operations or checkpoints. In normal scratch operations, the user moves his output and specific temp files manually, at some point before the purge, and leaves it to the purge mechanism to remove the rest of the temp files. Very few users are courteous to scratch space administrators and perform cleanup after their job completion. This obviously results in huge files occupying the scratch space unnecessarily. The purge mechanism will not delete them as they have just been created. With our approach, the purging of the temp files could now be specified in the eviction jobs. Consequently, removing temporary files from a run can be performed hand-in-hand with computation job completion. Finally, in order to capture the case where not all output data and temp files are explicitly specified by the user, the cache management performs a periodic LRU sweep with a large temporal window (similar to a purge, but with a significantly longer duration). Thus, with this mechanism, the scratch is truly used as a cache by removing the datasets that will not be used again, at least not in the near future.

4.2.3 Dataset Retention

In order to retain datasets in the scratch beyond the lifetime of a job run, we have introduced the `#Retain` directive in the job script. Using this directive, users can specify the datasets, the duration for which they need to be retained and their destination once evicted. In some cases, the output of one job can be input to another and the dependent task may not be scheduled until later. In such cases, it might be cost effective to retain the dataset in the scratch instead of evicting it to Tier 2 and populating it back into Tier 1 storage. This needs to be balanced with scratch space usage, particularly when there are many such co-dependent jobs. To address this, we submit a retention job that touches the datasets periodically to protect it from the purge mechanism and eventually evicts it after the retention period expires. To prevent users from artificially retaining datasets, we impose the restriction that the duration cannot be longer than the original scratch purge window. In essence, a retention job is an $Evict(n)$, where n is the duration the output dataset spends on scratch (Tier 1) beyond job completion. The degenerative case, $Evict(0)$ is an immediate eviction of result and temp files from the cache.

4.3 Discussion

A key observation of this work is that although individual users specify their job-specific constraints, the overall cache manager at the HPC center attempts to reconcile these with other jobs for globally efficient scratch management, with the aim to satisfy the three goals laid out in Section 2.1.

The cache manager analyzes all submitted jobs to arrive at a decision on which population jobs to launch at what time. The Tier 1 storage offers a finite amount of bandwidth to Tier 2, which is governed by the HPC center's connectivity. Consequently, the population and eviction jobs compete for the available bandwidth. It is conceivable that an eviction can interfere with a population job that needs to be completed in time so the computation can start. One can argue that evictions can wait as timely population of the input data determines job turnaround time. Although not implemented in the current prototype, one can imagine throttling eviction jobs by assigning them a lower priority compared to population jobs. On the flip side, delayed evictions result in unnecessary space consumption. Therefore, any prioritization of evictions needs to be balanced against available space.

In essence, our cache-based view of the scratch allows us to perform such optimizations if need be: to throttle certain parts of a workflow in order to achieve a higher degree of center-wide serviceability. The extant approach of manual, arbitrary data staging and offloading–or lack thereof–simply does not allow any such possibility.

5. LOG-DRIVEN SIMULATION

We have implemented the techniques described so far for managing the scratch space as a cache in a realistic simulator. The simulator is driven by nearly three years of job logs from the Jaguar supercomputer [3]. The logs contain each job's queue entry time, start time, predicted and actual wall time, the number of nodes needed for the job, and

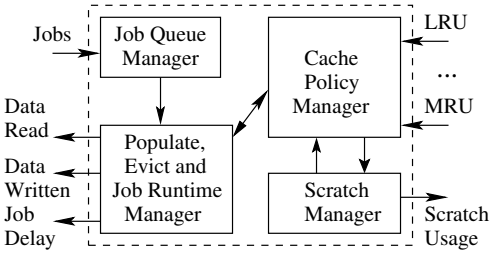


Figure 3: The architecture of the simulator.

the memory resources used by each node. Using this information the simulator models job queuing, scheduling, job start time, job execution times, and provides data about scratch space usage and the time it would take to stage and offload the required data for a given job. This information can then further be used to determine any delay in meeting job scheduling deadlines.

Figure 3 shows the main components of the simulator, namely, the job queue manager, the population, eviction, and job runtime manager, the cache policy manager, and the scratch manager. The job queue manager maintains all of the data collected from the logs, and determines when a job should begin the population process. The population, eviction, and runtime manager performs either a normal or a just-in-time staging and then schedules the job to run when resources are available. It also handles offloading of data. This module communicates with the cache policy manager to allocate scratch space for the job. The cache policy manager contains the implementations of the different caching mechanisms and is capable of determining the workflow information for a job. This module determines what datasets will be evicted out of the scratch space when the data needs of incoming jobs cannot be met. Once a job's data is selected for removal, the eviction module offloads it to secondary storage, and the scratch module is queried to free and allocate the associated space. In addition to modeling the scratch space the scratch module also provides accounting and statistics such as the scratch space used and the data read as well as other vital statistics.

We note that the total simulated scratch space capacity has no bearing on the simulation as we measure scratch utilization per hour, instead of cumulative utilization. Additionally, we also synthesize the job logs to introduce various dependencies and for testing usage scenarios. This is not an issue, because synthesizing job logs is a common practice when realistic dependency logs are unavailable. Moreover, the logs were randomly synthesized to be fair to all of the candidate techniques and do not favor our workflow-aware caching.

6. EVALUATION

In this section, we present an evaluation of our scratch as a cache approach using the simulator described in Section 5, driven by job-statistics logs collected over a period of three-years on the Jaguar [3] supercomputer. Table 1 shows some relevant characteristics of the logs.

In the following, we use our simulator to first justify the need for treating the scratch space as a cache, followed by an investigation of the various aspects of the caching model. In all of our experiments, we used 1TB as scratch capacity.

Table 1: Statistics about the job logs used in this study.

Duration	22764 Hrs
Number of jobs	80234
Job execution time	30 s to 120892 s, avg. 5835 s
Input data size	2.28 MB to 3714 GB, avg. 32.1 GB

6.1 Behavior of Traditional Caching Mechanisms

The goal of this set of experiments is to justify treating scratch space as a cache. For this purpose, we first study how scratch utilization is affected under the normal purge policies. Here, we set the purge period to seven days, and monitored the amount of scratch space used per hour. Figure 4(a) shows the results. In the beginning, the rate of job issue was not too high, so the periodic purge is able to keep the space utilization lower. However, as the rate of job issue increased, the scratch utilization became high. We note that having constant utilization is not undesirable, however, the after-the-fact purge will be unable to accommodate any instantaneous increase in job issue rate.

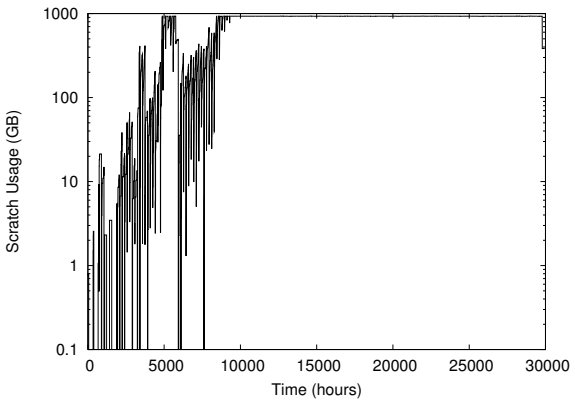
Next, we tried two different caching algorithms to manage the scratch: the commonly used LRU and MRU algorithms. We note that while LRU results in a steady increase in scratch utilization with an average of 20.1% higher usage per hour compared to 7-day Purge, MRU is able to drastically reduce scratch utilization per hour, with an average per hour savings of 98.4% compared to 7-day Purge. Thus, this result indicates that MRU would be a promising approach.

Additionally, from the timing of data read under the three approaches, as shown in Figure 5, we observe that under both 7-day Purge and MRU the data was staged much earlier than under LRU. This provides a different perspective in that the exposure window, E_w, under MRU is much greater than that under LRU, and thus indicates that MRU would be undesirable.

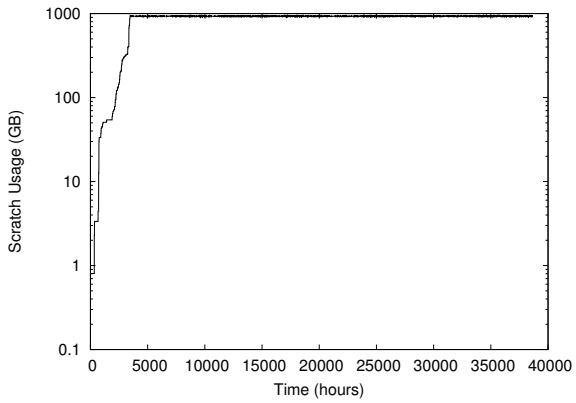
These results show that scratch utilization can be improved using better management, however, simplistic algorithms are unlikely to yield the desired objectives.

6.2 Effect of Workflow-Aware Caching on Scratch Utilization

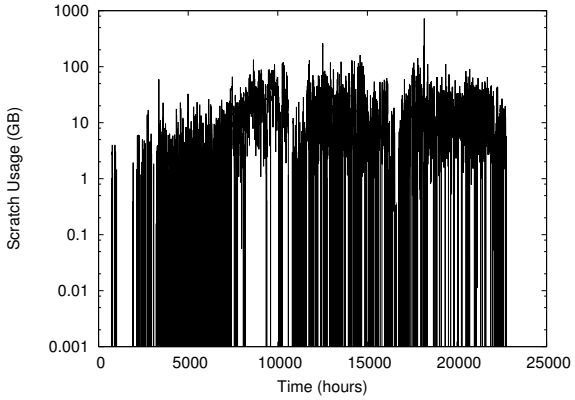
In this experiment, we first introduced job dependencies between the traced logs. Specifically, we synthesized the logs to introduce job dependencies of up to three jobs spread over a period of a week. In total, 59.8% of the jobs in the logs are chained into workflows. Then, we repeated the previous set of experiments with 7-day Purge, LRU, and MRU and also utilized our workflow-aware caching algorithm to study its affect. Figures 6 shows the data read, and Figure 7 shows the scratch utilization under our approach. We observe that workflow-aware caching was able to reduce the average scratch-space utilization per hour by 6.6% (e.g. 67.5 GB/Hr on average per Terabyte of storage) compared to LRU. Moreover, MRU read the most amount of data. This is because MRU blindly throws away the data as soon as it has been used, without any regard to whether it will be utilized again in the near future or not. In contrast, workflow-aware caching reduced the amount of data transferred when compared to 7-day Purge, LRU, and MRU by 1.8%, 5.7%, and 20.4%, respectively. Reducing the amount



(a) *7-day Purge.*



(b) *LRU.*



(c) *MRU.*

Figure 4: Average scratch utilization over the duration of the logs under traditional management and simple caching algorithms. All jobs are assumed to be independent.

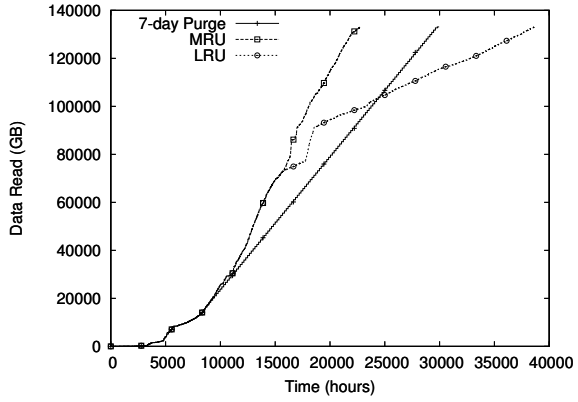


Figure 5: Data read under the studied approaches over time.

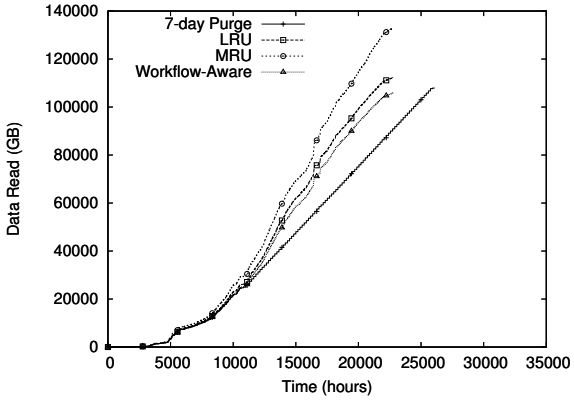
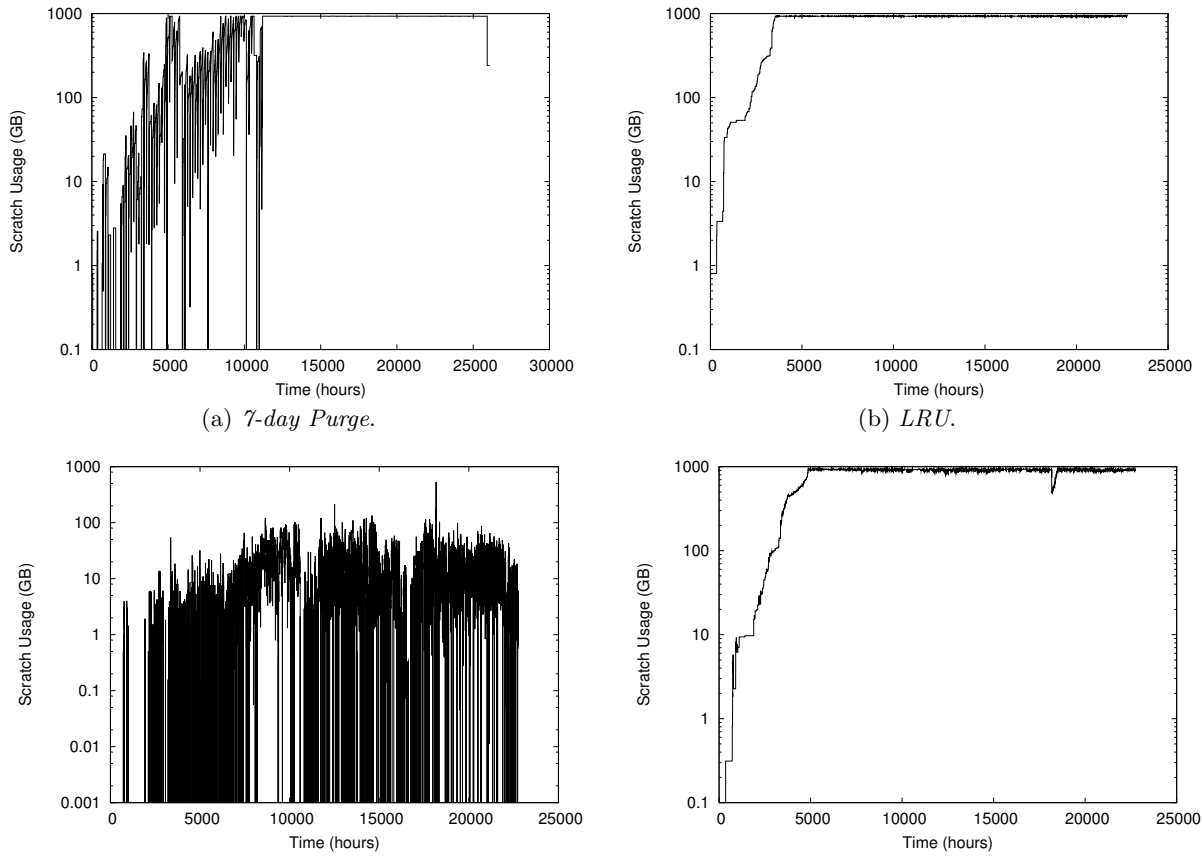


Figure 6: Data read under the studied approaches over time, when 59.8% of the jobs have workflow dependencies.



(a) *7-day Purge.* (b) *LRU.*

(c) *MRU.* (d) *Workflow-Aware Caching.*

Figure 7: Average scratch utilization over the duration of the logs under traditional management and simple caching algorithms. 59.8% of the jobs are chained in dependent workflows.

Table 2: Average expansion factor observed for the studied caching polices.

	7-day Purge	LRU	MRU	Workflow
Queue Entry	167289	2.54	2.59	2.54
Stage in Time	3.86	1.04	1.05	1.04

of data transferred also implies a lower probability of missing job scheduling deadlines due to smaller times required to bring all the necessary data into the scratch.

These results stress the need and the importance of integrating workflow information into scratch management.

6.3 Impact on Job Scheduling & Performance

We have shown that workflow-aware scratch management can improve space utilization and reduce data transfer requirements. In this set of experiments, we study the impact of the reduced data transfer on meeting job scheduling deadlines. In this context, system designers and funding agencies (e.g., US DOD, NSF, DOE, etc.) are adopting performance specification metrics such as *expansion factor* [13, 3] (EF), defined as the ratio *(wall_time + wait_time)/wall_time* averaged over all jobs (the closer to 1, the better). Therefore, we use expansion factor in our study.

Table 2 shows the EF for the studied approaches. First, we examined the queue entry time for determining the wait time in calculating the EF. Here, we observe that the traditional purge may lead to extremely high average EF as over time, the wait time accumulates as jobs are delayed as their data is staged in. Treating the scratch as a cache, reduces the EF to more acceptable values.

Next, we determined the wait time for our calculation using the time when staging for a job-associated data is initiated. This approach removes the accumulating delay affect and presents a more realistic EF. However, once again we observe that the traditional purge-based approach is far from ideal with 286.0% overhead, where as workflow-aware caching results in only 4.0% overhead.

Also note that, from these results it would seem that LRU is comparable to workflow-aware caching. However, we believe this to be an artifact of our job log synthesis when introducing workflow dependencies, and is not an argument for LRU-based caching being a suitable option in general for scratch management.

6.4 Effect of Types of Tier 2 Storage

In Section 3, we have discussed various storage devices that can act as Tier 2 storage for our scratch cache. In our next set of experiments, we study the affect of different types of Tier 2 on EF. Here, we consider our workflow-aware caching method, when the average Tier 2 Bandwidth is 10 Gbps, 250 Mbps, and 50 Mbps. We synthesized logs by randomly assigning a Tier 2 bandwidth to each job entry in the

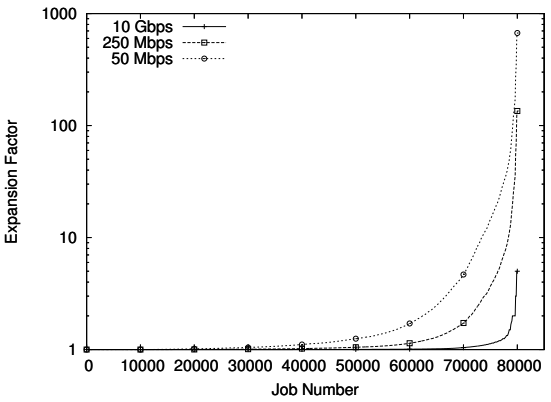


Figure 8: The distribution of EF as the average bandwidth to Tier 2 storage varies. 10 Gbps achieves a 1.04 average EF, while 250 Mbps achieves 2.09 and 50 Mbps achieves 6.51.

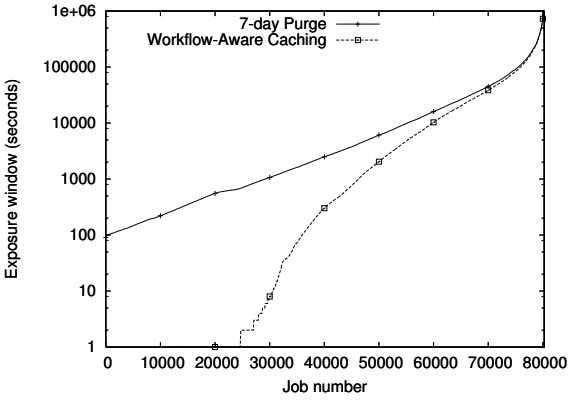


Figure 9: Distribution of observed exposure window under 7-day Purge and workflow-aware caching.

original log to denote input datasets originating from disparate data sources. Consequently, this results in a diverse staging in time.

Figure 8 shows the cumulative distribution of EFs under various Tier 2 average bandwidths. Even when all Tier 2 storage is capable of providing a low bandwidth of only 50 Mbps, 71.5% of the jobs finish with an EF less than 1.5 and an average expansion factor across all jobs of 6.51. Both 10 Gbps and 250 Mbps provide significantly better average EFs of 1.04 and 2.09, respectively.

6.5 Impact on Exposure Window

In the next experiment, we study how managing the scratch storage as a cache can help reduce the data exposure window, E_w. Figure 9 shows the distribution of exposure window under the 7-day Purge policy and our workflow-aware caching. It can be observed that workflow-aware caching is able to significantly reduce the exposure window, for 30.7% of the jobs, workflow-aware caching was effectively able to reduce E_w to zero, and for the remaining jobs it reduced E_w by 64.2%, i.e., 75.2% reduction on average across all jobs. Moreover, E_w was reduced by at least a factor of 10 for 48.3% of the jobs. Thus, workflow-aware caching is an effective means for reducing exposure of staged data to scratch failures.

In summary, the presented workflow-aware caching provides effective means to reduce average scratch utilization, reduces data that needs to be transferred per job, and allows for managing the scratch space in a globally optimal manner.

7. RELATED WORK

Several previous efforts address the coordination of data and computation activities in HPC centers. These range from simple dependency management in PBS [4] and Moab [17] to treating data activities as data jobs [14, 28]. However, our approach is a paradigm shift in how scratch storage is viewed and uses many of the aforementioned to realize a cache-based approach to HPC scratch management. In addition to synchronizing data movement using a suite of cache management tools, our work also addresses data retention, which only a workflow-aware caching scheme can accomplish.

Batch Aware Distributed File System (BAD-FS) [7] constructs a file system for large, I/O intensive batch jobs on remote clusters. BAD-FS addresses the coordination of input data and computation by exposing distributed file system decisions to an external workload-aware scheduler. We attempt to inherently improve the job workflow and center operations without creating a new file system, but by viewing the scratch as a cache.

Data staging and offloading tools such as Kangaroo [24], IBP [22] and other timely delivery tools [18, 19] provide standalone infrastructures to move data in and out of the HPC center. These tools can be used in conjunction with our cache population and eviction mechanisms.

A mature body of work, comprising of simple to advanced pattern-based approaches, exists for data caching [16, 8, 21, 12] and prefetching [21, 26, 6] to improve I/O performance and bridge the gap between the CPU and disk access speeds. In this paper, we exploit and leverage existing algorithms to better manage the scratch space.

Scientific data caches [25, 15] provide techniques to accelerate data accesses in the HPC center by offering dataset caching. However, these systems are not workflow-aware and perform simple LRU based cache replacement.

8. CONCLUSION

In this paper, we have argued for the need for treating the precious HPC center scratch space as a specialized cache that manages the inflow and outflow of necessary job data in a workflow-aware integrated fashion. We have presented the design and evaluation of a workflow-aware caching approach, which provides a 6.6% improvement in average scratch utilized per hour compared to an LRU based caching mechanism, and reduces the amount of data read on average by 9.3% compared to both a traditional purge and other caching approaches. Furthermore, the approach results in an improvement of 282.0%, on average, in the expansion factor – a popular metric to measure a center's serviceability – compared to the currently-used purging. Additionally, the presented approach works equally well for any kind of Tier 2 storage available to the users. Thus, our solution is able to reconcile several key factors such as reducing the duration of scratch space consumption, adapting to volatility,

and delivering the data on time. Finally, we note that the fundamental contribution of this work is the paradigm shift in managing the scratch space comprehensively and not as an after thought: this provides opportunities for HPC center managers to design customized scratch management as needed for their installations.

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, and by the U.S. National Science Foundation CAREER Award CCF-0746832.

9. REFERENCES

[1] GUPFS - Global Unified Parallel File System. http://www.nersc.gov/projects/GUPFS/, 2004.

[2] NCCS.GOV File Systems. http://info.nccs.gov/computing-resources/jaguar/file-systems, 2007.

[3] National Center for Computational Sciences. http://www.nccs.gov/, 2008.

[4] Pbs pro technical overview: Scheduling and file staging. https://secure.altair.com/sched_staging.html, 2008.

[5] Spider. http://www.nccs.gov/2008/06/30/nccs-launches-new-file-management-system/, 2008.

[6] S. Albers and M. Büttner. Integrated prefetching and caching in single and parallel disk systems. In *Proc. 15th ACM SPAA*, pages 24–39, Duluth, MN, June 2003.

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

[8] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

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

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

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

[12] C. Gniady, A. R. Butt, and Y. C. Hu. Program-counter-based pattern classification in buffer caching. In *Proc. USENIX OSDI*, 2004.

[13] Grid Infrastructure Group. TeraGrid, May 2007. http://www.teragrid.org/.

[14] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the grid. In *Proceedings of 24th IEEE International Conference on Distributed Computing Systems (ICDCS2004)*, 2004.

[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] N. Megiddo and D. S. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proc. 2nd USENIX FAST*, pages 115–130, San Francisco, CA, Mar. 2003.

[17] Cluster resources inc. http://www.clusterresources.com/, 2008.

[18] H. Monti, A. R. Butt, and S. S. Vazhkudai. Just-in-time staging of large input data for supercomputing jobs. In *Proc. PDS Workshop at SC08*, 2008.

[19] H. Monti, A. R. Butt, and S. S. Vazhkudai. Timely offloading of result-data in hpc centers. In *Proc. ACM ICS'08*, 2008.

[20] OpenPBS. Portable batch system. http://www.openpbs.org/.

[21] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. 15th ACM SOSP*, pages 79–95, Copper Mountain, CO, Dec. 1995.

[22] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Proceedings of the Network Storage Symposium*, 1999.

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

[24] D. Thain, S. S. J. Basney, and M. Livny. The kangaroo approach to data movement on the grid. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, 2001.

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

[26] N. Tran and D. A. Reed. ARIMA time series modeling and forecasting for adaptive I/O prefetching. In *Proc. 15th International Conference on Supercomputing*, pages 473–485, Sorrento, Italy, June 2001.

[27] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems*, 15(5):757–768, 1999.

[28] 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 *Proc. SC*, 2007.