An Integrated End-User Data Service for HPC Centers

Henry Matthew Monti

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Ali R. Butt, Chair
Wu-chun Feng
Heshan Lin
Calvin J. Ribbens
Sudharshan S. Vazhkudai

November 26, 2012
Blacksburg, Virginia, USA

Keywords: End-User Data Services, Scratch as a Cache, Data Offloading, Data Staging
Copyright 2012, Henry Matthew Monti

An Integrated End-User Data Service for HPC Centers

Henry Matthew Monti

(ABSTRACT)

The advent of extreme-scale computing systems, e.g., Petaflop supercomputers, High Performance Computing (HPC) cyber-infrastructure, Enterprise databases, and experimental facilities such as large-scale particle colliders, are pushing the envelope on dataset sizes. Supercomputing centers routinely generate and consume ever increasing amounts of data while executing high-throughput computing jobs. These are often result-datasets or checkpoint snapshots from long-running simulations, but can also be input data from experimental facilities such as the Large Hadron Collider (LHC) or the Spallation Neutron Source (SNS). These growing datasets are often processed by a geographically dispersed user base across multiple different HPC installations. Moreover, end-user workflows are also increasingly distributed in nature with massive input, output, and even intermediate data often being transported to and from several HPC resources or end-users for further processing or visualization.

The growing data demands of applications coupled with the distributed nature of HPC workflows, have the potential to place significant strain on both the storage and network resources at HPC centers. Despite this potential impact, rather than stringently managing HPC center resources, a common practice is to leave application-associated data management to the end-user, as the user is intimately aware of the application's workflow and data needs. This means end-users must frequently interact with the local storage in HPC centers, the *scratch space*, which is used for job input, output, and intermediate data. Scratch is built using a parallel file system that supports very high aggregate I/O throughput, e.g., Lustre, PVFS, and GPFS. To ensure efficient I/O and faster job turnaround, use of scratch by applications is encouraged. Consequently, job input and output data are required to be moved in and out of the scratch space by end-users before and after the job runs, respectively. In practice, end-users arbitrarily stage and offload data as and when they deem fit, without any consideration to the center's performance, often leaving data on the scratch long after it is needed. HPC centers 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 preselected time threshold called the purge window that commonly ranges from a few days to a week. This ad-hoc data management ignores the interactions between different users' data storage and transmission demands, and their impact on center serviceability leading to suboptimal use of precious center resources.

To address the issues of exponentially increasing data sizes and ad-hoc data management, we present a fresh perspective to scratch storage management by fundamentally rethinking the manner in which scratch space is employed. Our approach is twofold. First, we re-design

the scratch system as a "cache" and build "retention", "population", and "eviction" policies that are tightly integrated from the start, rather than being add-on tools. Second, we aim to provide and integrate the necessary end-user data delivery services, i.e. timely offloading (eviction) and just-in-time staging (population), so that the center's scratch space usage can be optimized through coordinated data movement. Together, these two combined approaches create our Integrated End-User Data Service, wherein data transfer and placement on the scratch space are scheduled with job execution. This 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 to serve a caching workload.

This work was sponsored in part by the LDRD program of ORNL, managed by UT-Battelle, LLC for the U.S. DOE (Contract No. DE-AC05-00OR22725), and by the U.S. NSF Awards CCF-0746832, CNS-1016408, and CNS-1016793.

Dedication

Dedicated to my parents, family, and friends for their encouragement and support.

Acknowledgments

First and foremost, I would like to thank my advisor Dr. Ali R. Butt. Ali is an extremely capable academic, who easily recognizes the problems that should be studied and the questions that need to be asked. Besides providing me with input and ideas, he pushed me to keep on working throughout this dissertation, and convinced me to continue on when I had doubts.

Next, I would like to thank Dr. Sudharshan S. Vazhkudai who has been our long time collaborator. His practical knowledge helped me understand how things really work at High Performance Computing facilities. Further, because of his efforts I was able to spend a summer at Oak Ridge National Laboratory, which was very enjoyable and enlightening, while also aiding this research.

I would also to thank Dr. Calvin J. Ribbens for motivating me to complete my research, as well as providing with me the opportunity to teach at Virginia Tech. Further, I would like to thank the rest of my dissertation committee: Dr. Wu-chun Feng and Dr. Heshan Lin for their insights and useful comments in pursuing this research and dissertation.

Moreover, I would like to thank my parents for their support and encouragement during the nearly six years I've taken to finish my Ph.D. It would have been much harder to complete my dissertation without the holidays at home and without all of the food I ended up taking back with me after visiting.

Finally, I would also like to thank my friends and colleagues at the Distributed Systems and Storage Laboratory (DSSL), especially M. Mustafa Rafique, Guanying Wang, Min Li, and Aleksandr Khasymski. We had quite a bit of fun at the many conferences we attended, and may have occasionally published some research.

Contents

1 Introduction **1**

1.1 Problem Statement 1

1.1.1 Challenges in HPC Scratch Management 2

1.1.2 Challenges in End-User Data Delivery 3

1.2 Research Objectives 5

1.3 Research Contributions 7

1.3.1 Scratch as a Cache 7

1.3.2 Timely Data Offloading 8

1.3.3 Just-In-Time Data Staging 8

1.3.4 Using the Cloud for End-user Data Delivery 8

1.4 Dissertation Organization 9

2 Background and Related Work **10**

2.1 Background on HPC Job Workflow 10

2.2 Related Work 11

2.2.1 GridFTP 11

2.2.2 IBP 11

2.2.3 BAD-FS 12

2.2.4 Kangaroo 12

2.2.5 Coordinated Data Movement 12

2.2.6 Peer-to-peer Data Transfer 13

2.2.7 The Network Weather Service . . . 14
2.2.8 Large System Reliability . . . 14
2.2.9 Timely Offloading and Staging . . . 14
2.2.10 Caching . . . 15

3 Rethinking HPC Center Scratch Storage **16**
3.1 Scratch as a Cache . . . 17
3.1.1 Rethinking "Hot" and "Cold" Contents . . . 17
3.1.2 Limitations of Current Scratch Management Techniques . . . 19
3.1.3 Cache Management Overview . . . 19
3.2 Workflow-Driven Caching . . . 20
3.2.1 Collecting Information from the Job Script . . . 20
3.2.2 Cache Operations . . . 21
3.2.3 Discussion . . . 24
3.3 Log-Driven Simulation . . . 24
3.4 Evaluation . . . 25
3.4.1 Behavior of Traditional Caching Mechanisms . . . 26
3.4.2 Effect of Workflow-Aware Caching on Scratch Utilization . . . 27
3.4.3 Impact on Job Scheduling & Performance . . . 29
3.4.4 Effect of Types of Tier 2 Storage . . . 29
3.5 Chapter Summary . . . 30

4 Timely Offloading for HPC Output Data **31**
4.1 Requirements of a Result-Data Offloading Service . . . 32
4.2 Design . . . 33
4.2.1 Architecture Overview . . . 34
4.2.2 Intermediate Nodes . . . 36
4.2.3 The Data Offloading Process . . . 38
4.2.4 Design Summary . . . 41

4.3 Implementation . . . 41
4.3.1 Integration with Job Submission System . . . 42
4.3.2 Integration with BitTorrent and NWS . . . 43
4.3.3 Deployment . . . 44
4.4 Simulating the Offloading Process . . . 45
4.5 Evaluation . . . 45
4.5.1 Implementation Results . . . 45
4.5.2 Simulation Results . . . 51
4.6 Chapter Summary . . . 55

5 Just-In-Time Staging of HPC Input Data 56
5.1 Design . . . 57
5.1.1 Architecture . . . 58
5.1.2 Supporting Timely Staging . . . 61
5.1.3 Discussion . . . 63
5.2 Implementation . . . 64
5.2.1 Integration with Job Submission . . . 65
5.2.2 Integration with BitTorrent and NWS . . . 65
5.2.3 Center-wide Global Staging Considerations . . . 66
5.2.4 Ensuring Data Reliability . . . 66
5.2.5 Multi-Input Staging . . . 66
5.3 Simulating HPC Data Staging Process . . . 67
5.4 Evaluation . . . 67
5.4.1 Implementation Results . . . 67
5.4.2 Log Analysis . . . 73
5.4.3 Simulation Results . . . 77
5.5 Chapter Summary . . . 79

6 A Cloud-Based Adaptive Data Transfer Service for HPC 80

6.1 Using the Cloud for End-User Data Delivery 81
6.1.1 Cloud as Intermediate Storage for Data Transport 82
6.1.2 Azure Data Services 82
6.2 Design 83
6.2.1 Design Overview 83
6.2.2 Cloud Data Interface 84
6.2.3 Data Transport as a File System 87
6.2.4 HPC Job Submission Integration 88
6.2.5 Viability of Using Cloud Resources 89
6.3 Implementation 89
6.3.1 Architecture 89
6.3.2 Data Operations 90
6.3.3 Real World Considerations 91
6.4 Evaluation 92
6.4.1 Implementation Results 92
6.4.2 Cost of Cloud Usage 96
6.5 Chapter Summary 96

7 Conclusion 98

7.1 Dissertation Summary 98
7.2 Future Research Directions 100

A Simulating HPC Data Management with simHPC 110

A.1 Job Scheduling 110
A.2 Trace-Driven Simulation 110
A.2.1 Job Traces 111
A.2.2 Bandwidth Traces 111
A.2.3 Simulator Output 111
A.3 Flow of Control in *simHPC* 112

List of Figures

1.1 Depiction of use-cases for a timely offload of result-data and just-in-time staging of input data. 5

3.1 (a) Traditional approach to scratch usage is a set of disjoint tools, performing uncoordinated 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. 18

3.2 An instrumented PBS script demonstrating the caching directives. 21

3.3 A single user job being parsed into population, computation, retention, and eviction jobs. 22

3.4 The architecture of the caching specific portions of the simulator. 25

3.5 Average scratch utilization over the duration of the logs under traditional management and simple caching algorithms. All jobs are assumed to be independent. 26

3.6 Data read under the studied approaches over time. 27

3.7 Data read under the studied approaches over time, when 59.8% of the jobs have workflow dependencies. 27

3.8 Average scratch utilization over the duration of the logs under traditional management and caching algorithms. 59.8% of the jobs are chained in dependent workflows. 28

3.9 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. 30

4.1 Depiction of usecases for a timely offload of result-data: (a) An expeditious offload to release center scratch space and to protect the data against a purge; (b) An end-user data delivery; and (c) Data delivery to another part of the job workflow. 33

4.2 Intermediate node discovery using random p2p messages. Here, the end-user submission site (black) discovers three intermediate nodes (gray). 37

4.3 The data flow path from the HPC center to the submission site. The intermediate nodes are represented by hexagons. The participants also run an instance of the NWS (gray square) for bandwidth monitoring. 39

4.4 Bandwidth monitoring using the NWS. 'S' indicates our software. 40

4.5 The per-node system components and their interactions. 42

4.6 An instrumented PBS script containing offload specific directives. 43

4.7 The experimental setup used for evaluation. 46

4.8 Utilized out-bound bandwidth at the center, as the system adjusts to failures and meets the 600s deadline for offloading. The labeled regions represent utilized bandwidth to individual nodes. 49

4.9 Available data under different error coding schemes, as intermediate nodes fail. 50

4.10 Overall scratch utilization over the duration of the traces under different approaches. The solid line shows the average utilization measured per hour. . . 51

4.11 The scratch savings achieved by using a decentralized offload compared to a standard 7 day purge. 52

4.12 Average Expanded Usage Factor for the HPC center for a duration of three years and 80,025 jobs, under a 7-day purge and decentralized offloading with erasure coding and two-copies for varying scratch sizes. 54

4.13 Average Expanded Usage Factor under decentralized offloading with replication and erasure coding for varying scratch sizes and different intermediate node failure rates. 55

5.1 Overview of our staging framework, and interactions between the components. 58

5.2 The data flow path from the client site to the HPC center. Each intermediate node (hexagon) runs NWS (gray square) for bandwidth monitoring. 62

5.3 Implementation architecture for timely staging. 64

5.4 An instrumented PBS script containing the directives for timely staging. . . 65

5.5 Configurations used in Multi-Input test. 70

5.6 Transfer time as different combinations of Level 1 (L1) and Level 2 (L2) nodes are failed. The results are normalized with respect to a direct transfer. . . . 71

5.7 The distribution of staging delay and re-transmission overhead for 25 transfers with one scratch space failure. `n` represents by how early data staging is started before job startup, with higher `n` implying an earlier start of the staging process. 72

5.8 The actual and user predicted run-times for each job in the logs. Users typically request significantly more wall time than necessary. 73

5.9 The R values for each job in the trace grouped into 10 bins. Each bin represents the fraction of the requested wall time actually used. 73

5.10 The effect of queue wait time on average input data size. Each bin value represents the maximum non-inclusive wait time for jobs in that bin. 74

5.11 The effect of queue wait time on utilization (node hour). Each bin value represents the maximum non-inclusive wait time for jobs in that bin. 74

5.12 The number of jobs in each bin. 75

5.13 The effect of queue wait time on average job run-time. 75

5.14 The effect of queue wait time on the average number of nodes used for a job. 75

5.15 The effect of queue wait time on average utilization (node hour). 75

5.16 Scratch savings under timely staging compared to direct transfers. The purge period is seven days. 78

5.17 The exposure window for each job in the log under different approaches. . . 78

5.18 The effect of batch queue prediction accuracy on the staging error. 79

6.1 The main software components of CATCH. 83

6.2 The data flow path from the HPC center to the end-user site. The intermediate resources are represented by hexagons. The gray squares represent software hooks/APIs that CATCH uses to control the data flow. 84

6.3 Different approaches for using the cloud to implement end-user data delivery services. 85

6.4 An example annotated job script with cloud storage specific directives. . . . 88

6.5 Architecture of CATCH. 90

6.6 Architecture of *Cloud FS*. 90

6.7 Transfer times (in seconds) to different cloud regions, using increasing numbers of streams. The file size used is 1 GB. 94

A.1 Control flow in *simHPC*. 112

List of Tables

3.1 Job script instrumentation directives for managing scratch as a cache. 21

3.2 Statistics about the job logs used by *simHPC*. 26

3.3 Average expansion factor observed for the studied caching polices. 29

4.1 Input parameters for the data offload manager. 34

4.2 New script directives used for offloading. 42

4.3 Inter-level bandwidth statistics. 45

4.4 Comparison of decentralized transfer times (in seconds) with different direct transfer techniques. The buffer size for IBP, GridFTP, and BBCP is set to 1 MB. The number of streams in GridFTP and BBCP is set to 8 and 16, respectively. 47

4.5 The time to transfer a 5.0 GB file using standard BitTorrent. The equivalent phases for our scheme are shown in brackets. 48

4.6 Relative improvement in file transfer times using BitTorrent under varying chunk sizes, compared to the default chunk size of 256 KB. 48

4.7 Statistics about the job logs used by *simHPC*. 51

4.8 Job delays under the different offloading approaches. 53

4.9 Observed job delays under decentralized offloading with erasure coding and two-copies, when 10, 25, or 50 percent of the first-level intermediate nodes have failed. 54

5.1 Average observed bandwidth between PlanetLab nodes during experimentation. All numbers are in Mb/s. 67

5.2 Comparison of decentralized transfer times with different direct transfer techniques. The buffer size for IBP, GridFTP, and BBCP is set to 1 MB. The number of streams in GridFTP and BBCP is set to 8 and 16, respectively. . 68

5.3 The time to transfer a 2 GB file using standard BitTorrent. The equivalent phases for our scheme are shown in brackets. 69
5.4 Comparison of multi-input data transfer under direct and decentralized staging. 70
5.5 Statistics about the job logs used by *simHPC*. 72
5.6 The number of jobs impacted by unexpected job failures. 76
5.7 Input data size and amount of queue wait time for jobs affected by the failure of other jobs. 77

6.1 Average observed bandwidth and transfer times for a 4 MB probe to different Azure regions. 93
6.2 The time to transfer a 1 GB file using multiple regions. 94
6.3 Comparison of decentralized transfer times (in seconds) with different direct transfer techniques. The buffer size for IBP, GridFTP, and BBCP is set to 1 MB. The number of streams in GridFTP, BBCP, and CATCH is set to 8, 16, and 16, respectively. 95
6.4 Current Azure pricing. 96
6.5 Cost of using CATCH for different workflows under varying pricing structure. 96

A.1 Statistics about the job logs used by *simHPC*. 111

Chapter 1

Introduction

1.1 Problem Statement

The advent of extreme-scale computing systems, e.g., Petaflop supercomputers, High Performance Computing (HPC) cyber-infrastructure, e.g., TeraGrid [19], Enterprise databases, and experimental facilities such as large-scale particle colliders, are pushing the envelope on dataset sizes. Supercomputing centers routinely generate and consume ever increasing amounts of data while executing high-throughput computing jobs. These are often result-datasets or checkpoint snapshots from long-running simulations, but can also be input data from experimental facilities. For example, the Jaguar petaflop machine [18] at Oak Ridge National Laboratory, which is No. 2 in the Top500 supercomputers as of this writing, is generating terabytes of user data while supporting a wide-spectrum of science applications in Fusion, Astrophysics, Climate and Combustion. Similarly, input datasets — from experimentation facilities such as the Large Hadron Collider (LHC) [43] or the Spallation Neutron Source (SNS) [15, 40] — are on track to reach petabytes of data [94, 10]. Moreover, these growing datasets are often processed by a geographically dispersed user base across multiple different HPC installations. End-user workflows are increasingly distributed in nature with massive input, output, and even intermediate data being transported to and from several HPC resources or end-users for further processing or visualization. This is a common use-case on the TeraGrid where result-data — from computations at any of the dozen sites nation-wide — are required to be delivered to the end-user. These TeraGrid sites host some of NSF's most powerful supercomputers such as Kraken [11] at the University of Tennessee, Ranger [17] at Texas Advanced Supercomputing Center and Blue Waters at the National Center for Supercomputing Applications, and executing distributed workflows on these and other HPC resources will of be continued importance.

The growing data demands of applications coupled with the distributed nature of HPC workflows, have the potential to place significant strain on both the storage and network

resources at HPC centers. Despite this potential impact, rather than stringently managing HPC center resources, a common practice is to leave application-associated data management to the end-user, as the user is intimately aware of the application's workflow and data needs. This means end-users must frequently interact with the local storage in HPC centers, the *scratch space*, which is used for job input, output, and intermediate data, that are currently on the order of terabytes. Scratch is built using a parallel file system that supports very high aggregate I/O throughput, e.g., Lustre [39], PVFS [37], and GPFS [91]. To ensure efficient I/O and faster job turnaround, use of scratch by applications is encouraged. Consequently, job input and output data are required to be moved in and out of the scratch space by end-users before and after the job runs, respectively. The scratch space requires proper provisioning to accommodate the storage demands of all incoming jobs, which in turn affects center serviceability. In practice, end-users arbitrarily stage and offload data as and when they deem fit, without any consideration to the center's performance, often leaving data on the scratch long after it is needed. HPC centers are aware of these constraints and enforce "purge" policies to manage the precious scratch space, wherein data is deleted based on a time window (ranging from a few hours to a few days) [5, 2]. As centers become crowded, the purge policies get more stringent to provide space for incoming jobs. The purge window is, therefore, a product of the center's load, its provisioned storage, and its desire to maintain a certain level of serviceability.

However, this ad-hoc approach to data management ignores the interactions between different users' data storage and transmission demands, and their impact on center serviceability. To address this, in this dissertation, we focus on comprehensive end-user data management, which has largely been marginalized under current compute-focused center provisioning policies. In the remaining sections of this chapter, we provide further discussion of the challenges posed by growing data demands and ad-hoc data management, as well describe the contributions made by this research while attempting to address those challenges. Specifically, Sections 1.1.1 and 1.1.2 describe the obstacles faced in scratch management and end-user data delivery, respectively. Section 1.2 details our objectives, while Section 1.3 outlines this dissertation's contributions. Finally, Section 1.4 describes the organization of the remainder of this dissertation.

1.1.1 Challenges in HPC Scratch Management

The scratch file system in an HPC center provides fast temporary local storage space for jobs, and is a precious commodity, 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 [39], PVFS [37], and GPFS [91]. 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 delivery 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 preselected time threshold called the 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 file storage. Instead, it is a special-purpose storage for the 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 the 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. 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.

1.1.2 Challenges in End-User Data Delivery

There are many challenges associated with building robust data transfer services for HPC. These services must provide high performance while transferring massive amounts of data to

and from a geographically distributed user-base with varied end-user connectivity, resource availability, and application requirements. Here we discuss the challenges of data delivery in an HPC environment and the limitations of the tools currently employed for offloading and staging data to a center's scratch space.

Data offloading With the exponential growth in application input and output data sizes, it is impractical to store all user data indefinitely. HPC centers are aware of this constraint and enforce purge policies to manage the precious scratch space by deleting data based on a time window (ranging from a few hours to days) [5, 2]. However, there is no corresponding end-user service for a timely offload of data, to avoid purging. This is largely left to the user and is a manual process, wherein users offload result-data using point-to-point transfer tools such as GridFTP [34], `sftp`, hsi [49], and `scp`. The inherent problem with several point-to-point transfer tools, used to offload data from supercomputers, is that they are only optimized for transfers between two well-endowed sites. For example, the TeraGrid offers several optimizations (TCP buffer tuning, parallel flows, etc.) for GridFTP transfers between the various site pairs that make up the TeraGrid, which are already well connected (10-40 Gbps links). In contrast, end-user data delivery involves providing access to the data at the user's desktop. How does one move data efficiently from well-provisioned HPC centers (e.g., Jaguar, Kraken [11], Ranger [17], etc.) or a cyber-infrastructure (e.g., TeraGrid) to the outside world? More often, users come from smaller universities and organizations with varied connectivity to the HPC center. Thus, efficient and timely delivery of data cannot be ignored as a "last-mile" issue.

Not providing a sophisticated solution for result-data delivery affects not only end-user service, but also center operations. The output data of a supercomputing job is the result of a multi-hour — even several days' — run. A delayed offload renders output-data vulnerable to center purge policies. The loss of output-data leads to wasted user time allocation that is very precious and obtained through rigorous peer-review. Thus, a timely end-user data offload can help optimize both center as well as user resources.

The need for such a service is also fueled by the, often, distributed nature of computing services and users' job workflow, which implies that data needs to be shipped to where it is needed. For example, several HPC applications analyze intermediate results of a running job, through visualizations, to study the validity of initial parameters and change them if need be. This process requires the expeditious delivery of the result-data to the end-user visualization application for online feedback. A slightly offline version of this scenario is a pipelined execution, where the output from one computation at supercomputer site A is the input to the next stage in the pipeline, at site B (Figure 1.1). Large-scale user facilities such as the Spallation Neutron Source (SNS) [15] and Earth System Grid (ESG) [4] that employ distributed workflows are already facing these problems and require efficient end-user data delivery techniques.

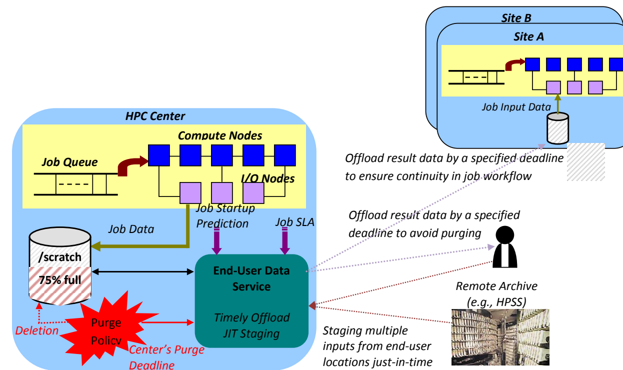


Figure 1.1: Depiction of use-cases for a timely offload of result-data and just-in-time staging of input data.

Data staging The inverse of delivering data to the end-user is to stage the data from a source location to an HPC center. Modern applications usually encompass complex analysis, which can involve staging large input data from observations or experiments. The data can originate from multiple sources ranging from end-user sites, remote archives (e.g., HPSS [44]), Internet repositories (e.g., NCBI [78], SDSS [94]), collaborating sites and other clusters that run pieces of the job workflow (e.g., Figure 1.1).

Once submitted, the job waits in a *batch queue* at the HPC center until it is selected for running, while the input data "waits" on the scratch space. HPC centers are heavily crowded and it is not uncommon for a job to spend hours — or even days — in the queue. Any failure during this wait time would entail expensive re-staging. In the best case when the data is staged at job submission, the input data spends the same time on the scratch as the job turn-around time, i.e., *(wall_time + wait_time)*. In the worst case, which is more common, the data waits longer as users conservatively (manually) stage it in much earlier than job submission. Thus, there is the need for an end-user data delivery service to stage the data just-in-time so it is able to minimize resource consumption and exposure of data to failure.

From the above use-cases, we can state the problem as: Offload by a specified deadline to avoid being purged; Or, Deliver by a specified deadline to ensure continuity in the job workflow. This naturally leads to the question of how to build end-user data delivery services that can be employed to mitigate the data delivery challenges in HPC.

1.2 Research Objectives

In this dissertation, we focus on comprehensive end-user data management, which has the potential to provide improved serviceability for HPC centers and quicker job turnaround for end-users. In light of the above discussions, we foresee several objectives for our Integrated End-User Data Service and this dissertation. Namely:

Reduce duration of scratch space consumption From a center standpoint, it is desirable to stage the input data of a waiting job as late as possible and to offload the output data from a completed job as early as possible, so that the scratch space is available for all of the currently running jobs' I/O (e.g., checkpointing and output). Thus, if a waiting or completed job's duration of scratch usage is reduced, it would help the HPC center better service the currently running jobs.

Reduce exposure to failures Another downside of leaving data on the scratch is its exposure to potential storage system failure. We refer to the time elapsed between when data is staged until the associated job starts running as the exposure window, E_w. To protect against storage failures, it is desirable to minimize E_w, preferably as close to 0 as possible. For example, supercomputers such as Jaguar, ASCI Q, ASCI White, and PSC Lemieux all cite storage as a primary reason for system downtime with MTBF of 37.5 hrs, 6.5 hrs, 40 hrs, and 6.5 hrs, respectively [52].

Timely delivery of HPC data One primary goal of this research is to deliver application data to center local storage from multiple sources just-in-time, or to end-users as quickly as possible after job completion. This must be accomplished in the face of both transient network conditions and changing batch queue job wait times or purge deadlines. Not properly accounting for such dynamism can have adverse effects for both the center and end-users: data delivery is delayed, and consequently job turnaround time is increased.

Minimize transfer times In addition to delivering data on time (either to the center or to the end-user), our framework should also attempt to minimize transfer times by choosing the best available routes and constantly reevaluating them. Reevaluation is critical to reacting to changes in the data delivery schedule in a timely fashion. For instance, this can help mitigate the effect of a sudden tightening in the delivery deadline due to the unexpected cancellation of a large job at the HPC center.

Avoid starvation and reduce job stall time Finally, from a center serviceability perspective, it is essential that the job scheduler not be rendered idle either because the input data of a waiting job has not been completely staged or because the waiting job's data has been purged from the scratch space prior to consumption. Such delays can result in jobs being stalled and possibly rescheduled — a costly process —, as they wait for the necessary input data to become available. Thus, comprehensive scratch management also requires utilizing techniques such as intelligent prefetching to stage job input data and selective retention of still useful data.

1.3 Research Contributions

To address the issues of exponentially increasing data sizes and ad-hoc data management, this dissertation presents a fresh perspective to scratch storage management by fundamentally rethinking the manner in which scratch space is employed. Our approach is twofold. First, we re-design the scratch system as a "cache" and build "retention", "population", and "eviction" policies that are tightly integrated from the start, rather than being add-on tools. Under the "Scratch as a Cache" paradigm, end-users' data must be moved into the cache from remote sources such as repositories or experimental facilities, processed at the center, and the resulting data must then be evicted to remote sites for analysis. In reality, population and eviction are large data transfers that correspond to data staging and data offloading, respectively. As a result, the second component of our approach is to provide and integrate the necessary end-user data delivery services, i.e. timely offloading (eviction) and just-in-time staging (population), so that the center's scratch space usage can be optimized through coordinated data movement.

In the remainder of this section, we highlight the specific contributions made by each component in our approach. Together these combined strategies enable the creation of our Integrated End-User Data Service.

1.3.1 Scratch as a Cache

We present a novel method [73] for managing an HPC center's scratch space, where the scratch is treated like a cache. This approach limits unnecessary direct end-user interaction, and allows caching policies to manage the scratch based job workflow needs, rather than end-users' desires. To facilitate this approach, we build 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. With this information, we can move beyond just a better purge policy to create integrated data services that work as fundamental part of the job workflow. These services can stage remote data to the scratch just before it is needed (population), offload output data to end-users after job completion (eviction), while retaining input, intermediate, and output data that will be reused again in the near future. This strategy 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.

1.3.2 Timely Data Offloading

We design a combination of both a staged as well as a decentralized offloading scheme [70, 72, 76] for job output data, which makes use of distributed intermediate sites. Compared to a direct transfer, our techniques have the added benefits of resilience in the face of end-resource failure and the exploitation of orthogonal bandwidth that might be available in the end-to-end data path. Additionally, we develop a decision making component that factors in parameters such as a center's purge deadline, the user delivery schedule, and a snapshot of current network conditions between the center and the end-user, to determine the most suitable approach to offload. We employ active monitoring, using the Network Weather Service (NWS) [107], to make the data offload process react to bandwidth degradation, thus ensuring that a user-specified delivery constraint or a purge deadline can be met. Finally, we have developed our solution in the context of real-world tools such as PBS [32] job submission system and BitTorrent [41].

1.3.3 Just-In-Time Data Staging

We present a JIT staging framework [71, 74, 77] that attempts to have the data available at scratch, from multiple input sources, just before the job is about to run. The framework proactively brings the data to intermediate storage sites on the path from the end-user site to the HPC center. This reduces the time for copying the data to scratch, thus providing better opportunities for JIT staging. JIT staging faces the additional constraint of volatile job startup times. To address this, we make use of Batch Queue Prediction [7] which provides estimates of job startup times. Furthermore, we employ an innovative combination of high-efficiency data dissemination (BitTorrent [41]) and network monitoring (NWS [107]) to exploit orthogonal, residual bandwidth and to dynamically adapt to network volatility, respectively, to improve overall scratch utilization.

1.3.4 Using the Cloud for End-user Data Delivery

We provide a cloud storage framework [75] for HPC, which utilizes proactive staging and offloading of data to cloud storage locations so as to have the input data available at the scratch storage — from multiple input sources — just before the job is about to run, and to offload output data — from scratch to the cloud — as soon as the job completes. Our framework is integrated with cloud resources exported by Windows Azure [67]. We adopt a novel variation to the use of intermediate nodes that differs from how they are used in most decentralized systems. The nodes participating in the transfer are in fact cloud resources, with specified reliability guarantees, thereby eliminating the fundamental concern of data delivery through a set of unreliable nodes. We have exported our end-user data delivery service through the file system abstraction provided by FUSE [22]. End-user programs can thus write and read to cloud storage and move data through them using standard file system operations.

1.4 Dissertation Organization

The remainder of this dissertation presents our Integrated End-User Data Service and is organized as follows. In Chapter 2, we discuss necessary background information and relevant related work in the field of HPC data management. In Chapter 3, we present our Scratch as a Cache model, and demonstrate how treating the scratch as a cache has the potential to improve the center's overall serviceability. In Chapter 4, we examine the challenges faced, and the strategies our approach employs in building a robust data offloading service, which can offload end-user data prior to a purge deadline. In Chapter 5, we build on our work in Chapter 4 and provide techniques and methods that allow end-user input data to be staged to the center scratch space just-in-time. In Chapter 6, we investigate using cloud resources as part of a collaborative end-user data delivery service. Additionally, we describe how cloud storage can play an important role in HPC data management. Finally, we conclude this dissertation and discuss future research directions in Chapter 7.

Chapter 2

Background and Related Work

2.1 Background on HPC Job Workflow

In this section, we present a brief primer on the current methodologies adopted in executing jobs at an HPC center. Users typically write proposals to obtain compute time at HPC facilities. These proposals are rigorously peer reviewed and, consequently, the allocated compute time is a valuable commodity. As we mentioned earlier, long-running user jobs typically use datasets from many sources. These datasets are required to be staged on scratch for analysis and the outputs are moved back to the end-user locations. HPC centers are equipped with various storage systems for different user needs: *home* is meant for development and compilations and is served using a network file system; *scratch* is for high-speed I/O of running jobs and is served using parallel file systems, e.g., Lustre [93]; and, *archives* are for long-term storage and only support lower interaction rates. Thus, users are encouraged to stage their input data onto the scratch for higher throughput and job turnaround.

Users usually submit their job requirements using scripts submitted to the scheduler at the center. Users either stage their data manually, in an out-of-band fashion, or include the staging commands in the scripts. Manual staging is error-prone and lacks coordination with job start-up times. Scripted staging wastes compute allocation as it is performed as part of the compute job; allocated cores are waiting while the data is being staged. In either case, point-to-point data movement tools, e.g., secure copy (`scp`), GridFTP [29], hsi [49], etc. are used to move data. As mentioned previously, these point-to-point transfer tools are only optimized for transfers between two well-endowed sites, and they require the end-user to be available for the duration of the time consuming data transfer process.

Once submitted, the job waits in a *batch queue* at the HPC center until it is selected for running, while the input data typically "waits" on the scratch space. In the best case when the data is staged at job submission, the input data spends the same time on the scratch as the job turn-around time, i.e., *(wall_time + wait_time)*. In the worst case, which is more

common, the data waits longer as users conservatively (manually) stage it in much earlier than job submission. As mentioned earlier, staging in the script or as part of the job run does not solve this problem because while data may spend less time on the scratch space, precious compute resources are wasted. Thus, the fundamental research problem of orchestrating data movement and computation remains open and is the focus of this dissertation.

2.2 Related Work

HPC data management is a critical research area, and a number of works have explored it from different perspectives. In the following, we discuss several related works.

2.2.1 GridFTP

GridFTP [34], is a high performance extension to the FTP protocol, which provides authentication, parallel transfers, and allows TCP buffer size tuning. The GridFTP overlay network service [88] implements a specialized data storage interface (DSI) to achieve split-TCP functionality. The GridFTP client command is issued with source and destination URLs A/C and C/D to denote a transfer between end points A and D through nodes C and D. In [54], the authors have extended this effort to use previous transfers as a measure of the performance of a particular node in the transfer overlay. These approaches address a similar goal of improving transfers through the use of orthogonal bandwidth between two sites. Other work on history based GridFTP transfer predictions [103] suggested that the relatively large error in predictions can be reduced through the factoring in of dynamic measurements. Our work differs as it focuses on making scratch space consumption more efficient through caching policies and coordinated data movement. Our specific techniques enable the delivery of data by a user-specified deadline or job start time, and further use dynamic measurements to adapt and adjust the fan-out of transfers. However, the GridFTP transfer protocol (or the other protocols described in this chapter) could be used as part of our framework in place of our timely offloading and just-in-time staging services.

2.2.2 IBP

IBP [83] offers a data distribution infrastructure with a set of strategically placed resources, storage depots, to move data. Together with the transport protocol, this is referred to as logistical networking, and a network of storage depots are being built along major national testbeds such as Internet2 [9], ReDDNet [6], etc., to enable end-user data delivery.

Our approach also exploits the presence of pre-installed storage nodes for data delivery as and when they are available. However, the main difference between IBP and our approach is

that instead of only relying on specialized resources, we leverage general-purpose resources to achieve end-user data delivery. Moreover, we differ in our approach to combine both a staged as well as a decentralized data delivery. The induction of user-specified nodes also allows the system to optimize the offload or stage-in on a per-user basis, which is not possible with IBP. Further, our approach is unique as we strive to meet a deadline in data delivery and stage in or offload to/from the HPC center.

In [35], the authors stream outputs from GTC runs through logistical networking. The adaptive buffer strategy reconciles the rate of data production with that of available network resources by failing over the transfer to a local IBP depot in case of a network failure. The goal here is to overlap computation with in-situ network data transfer. This is complementary to our work and such in-situ processing can also benefit from our decentralized transport.

2.2.3 BAD-FS

Batch Aware Distributed File System (BAD-FS) [33] 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.

2.2.4 Kangaroo

The use of intermediate buffers to hide latency or to provide fault tolerance is a common practice in OS as well as file systems. Kangaroo [100] extends this idea to Grid computing, with the goal to provide reliability against transient resource availability. It hides network storage using an application perceived file system with relaxed consistency semantics. The primary goal of Kangaroo is to provide reliability for Grid data transfers in the face of transient resource availability. However, Kangaroo simply provides a staged transfer mechanism and does not concern itself with network vagaries or changing route dynamics in an end-to-end data path.

2.2.5 Coordinated Data Movement

Several previous efforts address the coordination of data and computation activities in HPC centers. These range from simple dependency management in PBS [13] and Moab [69] to treating data activities as data jobs [57, 108]. However, our approach is a paradigm shift in how scratch storage is viewed and uses many of the aforementioned techniques 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.

Stork [57] a scheduler for data placement activities in a grid environment, along with Condor [63] and DAGMan [45], is used to schedule data and computation together in the face of vagaries. However, these systems are positioned as a part of the application workflow rather than a set of HPC center integrated services, where our work resides. Further, our approach provides a way to deliver data in time through the exploitation of multiple data flow paths in a resilient fashion.

DMOVER [8] is a tool that is used for moving data in the TeraGrid by aggregating data transfer commands in a script and scheduling them using a separate queue. However, it only addresses point-to-point data transfers using GridFTP. stagesub [108], that is deployed on Jaguar, Kraken and at LBL addressed coordinating data staging and offloading alongside computation through the use of data queues and job dependencies. However, it does not address JIT staging, volatility and delivery based on a deadline. Our work can be used in conjunction with these efforts to schedule a decentralized data delivery to coincide job completion. Our solution can also be built upon to transfer intermediate checkpoint data as long as our system is notified about the availability of the data.

2.2.6 Peer-to-peer Data Transfer

A number of systems such as Bullet[59, 58], Shark [30], and CoBlitz [79] have explored the use of multicast and p2p-techniques for transferring large amounts of data between multiple Internet nodes. The focus of these systems is on downloading of user data, or receiving multimedia streams. Our end-user delivery services require factoring in center-user service agreements and dynamic cloud resource availability, which are not considered in these systems. Content distribution networks (CDN) such as CoDeeN [105] effectively implement a system of proxy servers that users can explicitly use for faster delivery of data to their nodes. Large files in CoDeeN are transferred using multicast streams. Similarly, FastReplica [38] creates replicas of data on different content distribution nodes to support faster data access. Our work shares with these systems the goal of utilizing multiple paths for transferring large amounts of data, but differs in its focus on HPC applications and automatic dynamic selection of intermediate nodes to facilitate multicast when necessary.

A number of bulk data-transfer protocols have been developed for Internet use, e.g., Slurpie [96] allows clients to simultaneously contact a server and use random back-off to avoid performance degradation due to congestion. The approach of downloading large files from several mirror sites has been validated by its wide-spread use in BitTorrent [41], and many protocols for parallel downloading from mirror sites have been proposed [89, 86, 42]. However, these techniques have not been applied to large, scientific data in an HPC context (several orders of magnitude larger than p2p data sizes) and are also not aware of application-level delivery constraints [55]. These works are complimentary, and we built on the principles developed in these systems.

2.2.7 The Network Weather Service

The Network Weather Service (NWS) [107] provides a powerful framework which allows the resources of distributed computers to be monitored. A number of different resources can be observed such as the pair-wise bandwidth between computers and each computer's CPU utilization, though there are many other options. NWS bandwidth measurements have been used in a static context to determine a Grid data site, offering optimal download rates, from among multiple replicated alternatives [104, 103]. In this dissertation, however, we use measurements to determine a path within a network of nodes and dynamically adjust it based on bandwidth degradation.

2.2.8 Large System Reliability

Moving data away from a centralized storage, such as HPC centers, is also encouraged by recent studies that show that the rate of storage system failures is high [92, 82, 95]. Previous works [31, 53, 87] tried to improve reliability in large-scale installations, such as the HPC scratch space. These works entail going through a rigorous and time-consuming acquisition process mired with delays. In contrast, the use of intermediate and cloud storage nodes can provide a solution that can be arbitrarily grown to accommodate any desired level of storage while providing scalability and reliability.

2.2.9 Timely Offloading and Staging

Timely offloading of output data from the HPC center can only be achieved by coinciding the output data movement with the completion of the compute job. Previous work in this regard treats data offloading as an I/O job and schedules it alongside computation so it begins at job completion [108]. Such a coordinated approach optimizes center resource usage and ensures timely data delivery. Techniques presented in this dissertation complement the coordinated scheduling and the two approaches can be used together.

Staging of data on the HPC center to enable timely execution of jobs is another related direction. Recent work on staging by others [27] as well as our own [73, 71] has shown the importance of integrating staging services into center management software. Such works are complementary to our work, and vice versa, in that an integrated staging and offloading solution can significantly improve the overall serviceability of a center.

2.2.10 Caching

A mature body of work, comprising of simple to advanced pattern-based approaches, exists for data caching [66, 50, 61, 60] and prefetching [80, 102, 28] to improve I/O performance and bridge the gap between the CPU and disk access speeds. In this dissertation, we exploit and leverage existing algorithms to better manage the scratch space.

Finally, scientific data caches [101, 64] 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.

Chapter 3

Rethinking HPC Center Scratch Storage

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 the very large input, output, and intermediate data of currently running and soon-to-run user jobs. Developers are encouraged to use the scratch so their applications benefit from its high performance and abundant storage space. Moreover, the scratch is seldom regulated with quotas to avoid introducing any ceilings on applications' data sizes. However, the scratch space is not intended to be used as a generic file system for persistent user file storage. Instead, it is a special-purpose storage for the needed ("hot") data of running and waiting jobs. Nonetheless, in practice, the scratch space is utilized as a traditional file system, with purge polices added as an afterthought to delete the unneeded ("cold") data of finished jobs and to cap the scratch utilization within limits. This ah-hoc data management leads to sub-optimal use of scratch space, which should be used for new incoming or currently running jobs.

The dearth of comprehensive scratch space management results in significant drawbacks for both centers and end-users. Users arbitrarily stage and offload data as and when they deem fit, without any consideration to the center performance. 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. Resources used for provisioning the scratch space cannot be used to buy more compute power. Efficiently managing a center's scratch space can transform the productivity of even an under-provisioned scratch storage system.

In this chapter, we argue that re-envisioning the scratch as a cache captures the current HPC usage pattern more accurately than simple purge policies and better equips the scratch storage system to serve growing HPC datasets, while addressing the concerns discussed above. To facilitate this approach, we design cache retention and eviction policies are tightly inte-

grated from the start, rather than being add-on tools. Additionally, we present a work-flow aware caching approach that uses "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. Moreover, we validate our approach through simulation, comparing it against standard scratch management techniques. This approach allows application developers and end-users to make use of high performance scratch resources, while also limiting unnecessary direct end-user interaction. It allows caching policies to manage the scratch based job workflow needs, rather than end-user desires.

3.1 Scratch as a Cache

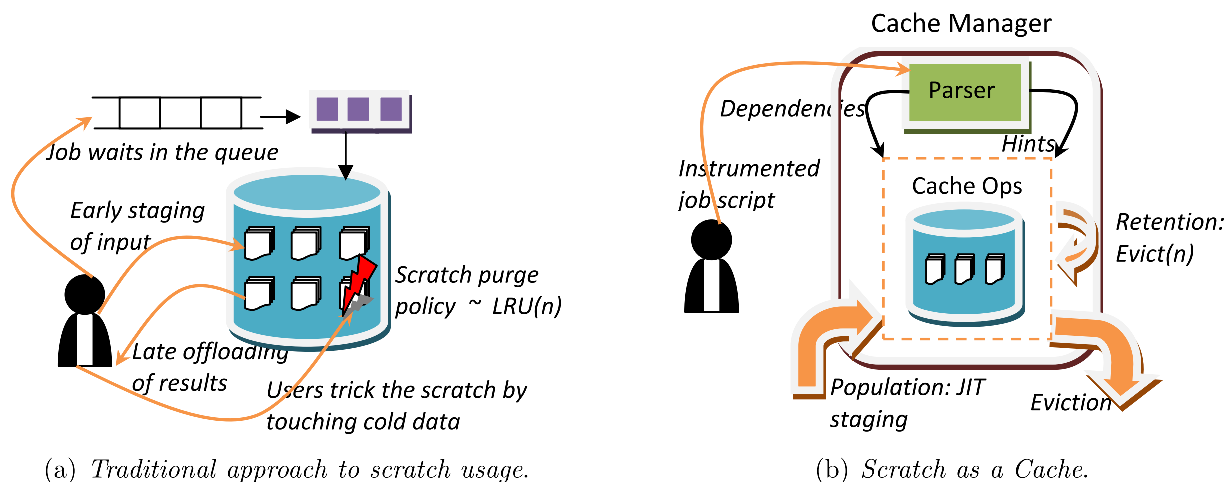
The key idea behind our approach is to view the HPC scratch storage as a cache. To this end, we need to support three basic cache operations namely, "populating" the cache, "retaining" appropriate datasets in the cache, and "evicting" datasets from the cache. Further, we must be able to determine the scenarios where HPC data is "hot" and should be populated or retained within the cache, and determine when HPC data is "cold" and therefore should be evicted 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.

3.1.1 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 the 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) *Traditional approach to scratch usage.* (b) *Scratch as a Cache.*

Figure 3.1: (a) Traditional approach to scratch usage is a set of disjoint tools, performing uncoordinated 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.

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 retouching. 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 with 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. In the next section, we further discuss the limitations of traditional scratch management.

3.1.2 Limitations of Current Scratch Management Techniques

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, 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 [5]. However, this ad-hoc approach is unable to capture many of the discussed scratch usage scenarios highlighted earlier. 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 the uncoordinated job submission process. Figure 3.1(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 for higher performance. At the other end, leadership-class facilities that boast 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.

3.1.3 Cache Management Overview

Figure 3.1(b) presents an overview of scratch as a cache. In this approach, direct user managed operations on the scratch storage are 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 to failures. 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 3.1(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 [16, 1], archives [44] at the center, user-specified nearby storage [83, 72], or end-user locations. Data is moved into and out of Tier 1 from/to Tier 2 storage using cache management tools.

3.2 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 the 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 uncoordinated 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.

3.2.1 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.

```
#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
```

Figure 3.2: An instrumented PBS script demonstrating the caching directives.

Instrumenting the Job Script

To support cache management, we have instrumented the PBS [32] 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. Figure 3.2 shows 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. Further, Table 3.1 shows a summary of the directives supported in our instrumented job scripts.

3.2.2 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

Directive	Function
Populate	Move job input data into the scratch from Tier 2 storage, but only when necessary
Evict	Move job output and intermediate data out of the scratch and to Tier 2 storage
Retain	Essentially $Evict(n)$, where n is the retention period for the dataset

Table 3.1: Job script instrumentation directives for managing scratch as a cache.

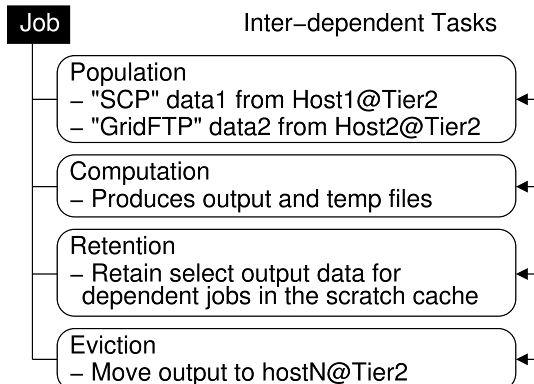


Figure 3.3: A single user job being parsed into population, computation, retention, and eviction jobs.

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 3.3). To this end, we leverage work, such as [108], which exploit modern resource manager (e.g., PBS [32], Moab [69]) 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.

Populating the Cache

The cache management suite 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 staging (Chapter 5) to bring the data in to coincide with job startup and attempt to minimize the 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 [107]), 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 software 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.

Evicting from the Cache

Evicting the output data of a job is essentially offloading it to Tier 2 storage. The transfer protocol and authentication to be used for this operation are provided as hints in the `#Evict` directive in the job script. The eviction job is configured to begin immediately after completion of the compute job. In addition to moving the result output data, the user can also identify "temp" files from 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 needed 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, purging the temp files can 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 software 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.

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 the 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.

3.2.3 Discussion

An important observation of this work is that although individual users specify their job-specific constraints, the cache management suite at the HPC center attempts to reconcile these with other jobs for globally efficient scratch management, with the aim to satisfy the goals laid out in Chapter 1.

The cache management software analyzes all submitted jobs to make a decision regarding 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.

3.3 Log-Driven Simulation

We have implemented the techniques described so far for managing the scratch space as a cache as part of a realistic simulator, *simHPC*, described in Appendix A. In this section, we describe the caching specific components added to *simHPC*. The simulator is driven by nearly three years of job logs from the Jaguar supercomputer [18]. 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 the memory resources used by each node. Using this information the simulator models job queuing, scheduling, job start times, 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.

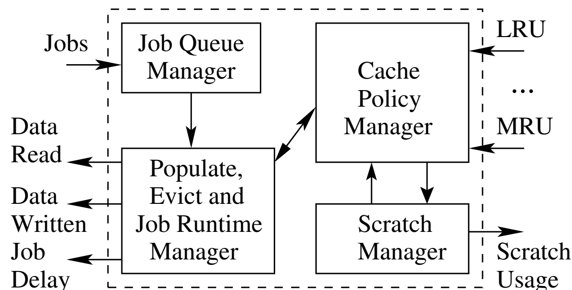


Figure 3.4: The architecture of the caching specific portions of the simulator.

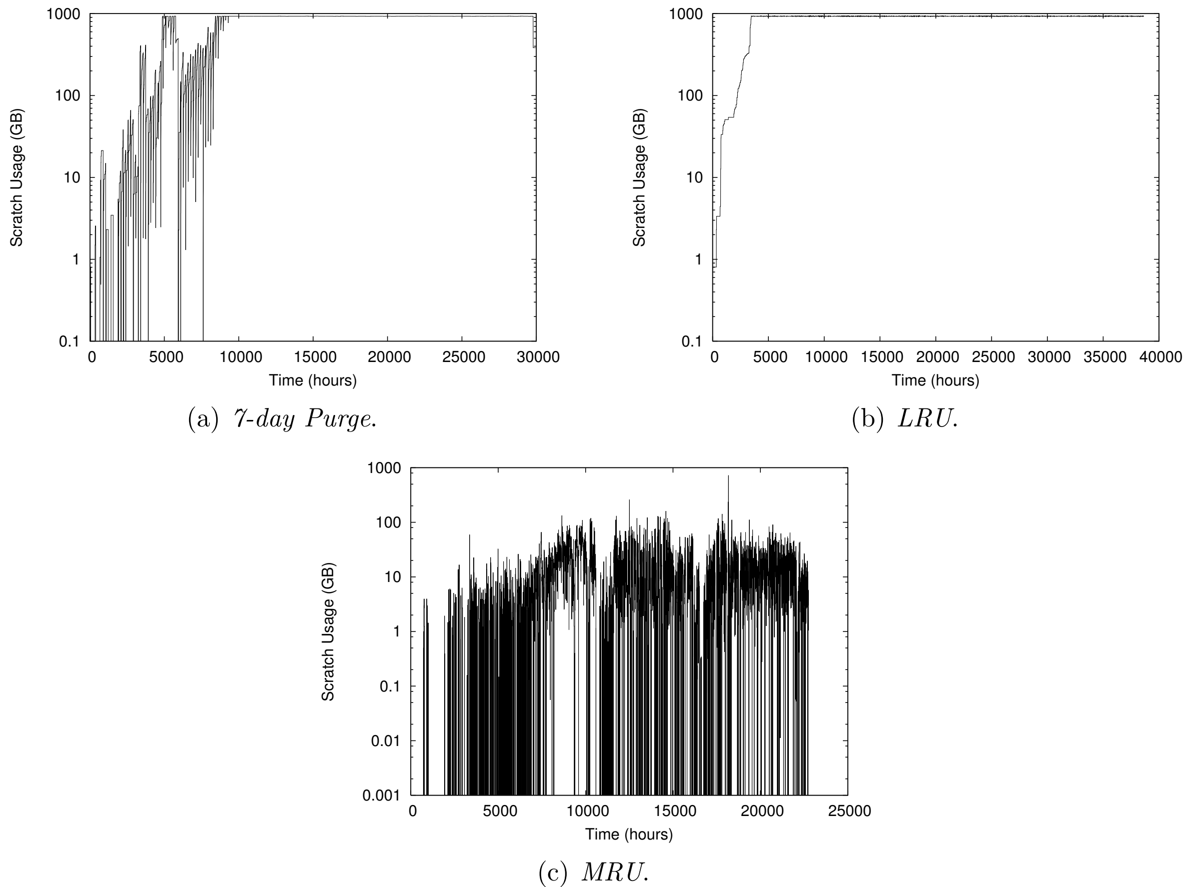
Figure 3.4 shows the additional components built on *simHPC*, namely, the job queue manager, the population, eviction, and job runtime manager, the cache policy manager, and the scratch manager. The job queue manager maintains all of the data collected from the logs, and determines when a job should 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.

3.4 Evaluation

In this section, we present an evaluation of our scratch as a cache approach using *simHPC*, described in Section 3.3 and in Appendix A. *simHPC* is driven by job-statistics logs collected over a period of three-years on the Jaguar [18] supercomputer. Table 3.2 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 1 TB as scratch capacity.



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

(c) *MRU.*

Figure 3.5: Average scratch utilization over the duration of the logs under traditional management and simple caching algorithms. All jobs are assumed to be independent.

3.4.1 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 3.5(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

Table 3.2: Statistics about the job logs used by *simHPC*.

Duration	22753 Hrs
Number of jobs	80025
Job execution time	1 s to 120892 s, average 5849 s
Input data size	2.28 MB to 7481 GB, average 65.3 GB

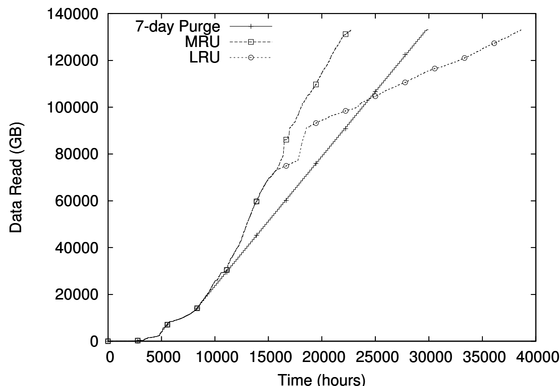


Figure 3.6: Data read under the studied approaches over time.

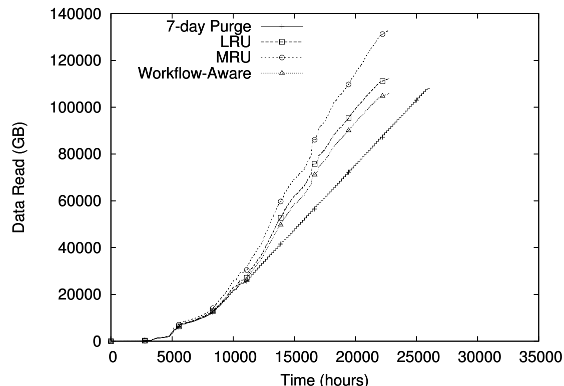


Figure 3.7: Data read under the studied approaches over time, when 59.8% of the jobs have workflow dependencies.

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.

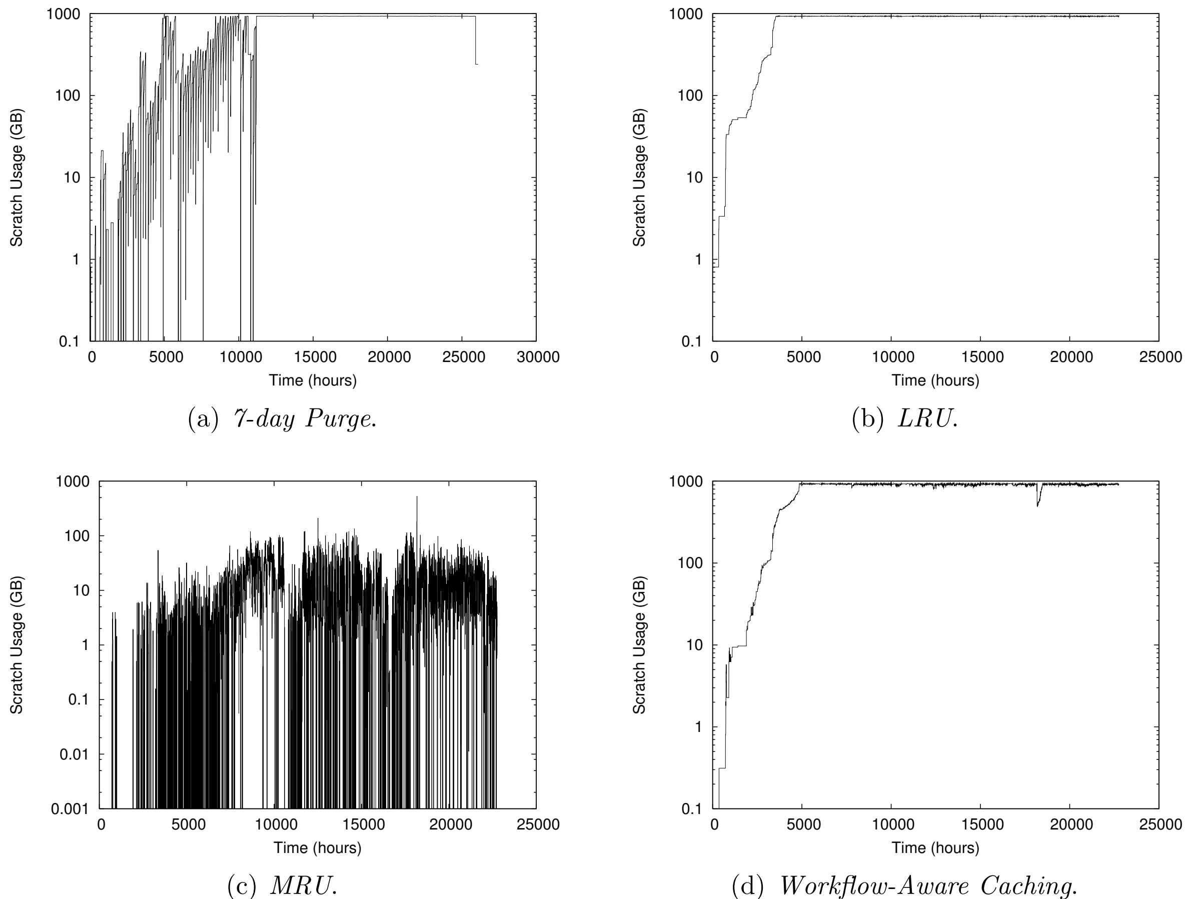
Next, we tried two different caching algorithms to manage the scratch: the commonly used LRU (Figure 3.5(b)) and MRU (Figure 3.5(c)) algorithms. We note that while LRU results in a steady increase in scratch utilization with an average of 20.1% higher usage per hour compared to 7-day Purge, MRU is able 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, Figure 3.6, shows the amount of data read under the three approaches. Here, we observe that under both 7-day Purge and MRU the data was staged much earlier than under LRU, although the same amount of data is eventually read under all the three schemes. 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.

3.4.2 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 effect. Figures 3.7 shows the data



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

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

Figure 3.8: Average scratch utilization over the duration of the logs under traditional management and caching algorithms. 59.8% of the jobs are chained in dependent workflows.

read, and Figure 3.8 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 reads 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 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.

3.4.3 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* [19, 18] (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 3.3 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.

3.4.4 Effect of Types of Tier 2 Storage

In Section 3.1, 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 original log to denote input datasets originating from disparate data sources. Consequently, this results in a diverse staging in time.

Table 3.3: 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

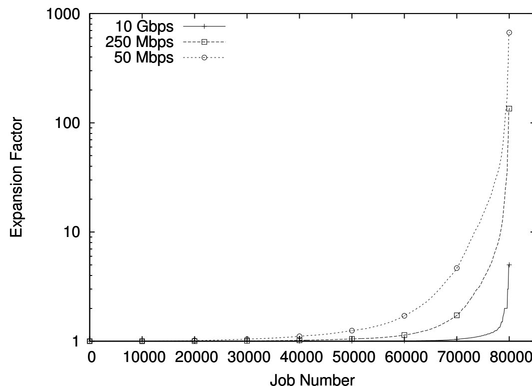


Figure 3.9: The distribution of EF as the average bandwidth to Tier 2 storage varies. 10 Gbps achieves a 1.04 average EF, while 250 Mbps achieves 2.09 and 50 Mbps achieves 6.51.

Figure 3.9 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.

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.

3.5 Chapter Summary

In this chapter, we have argued for treating the 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 an improvement in average scratch utilized per hour compared to an LRU based caching mechanism, and reduces the amount of data read on average compared to both a traditional purge and other caching approaches. Furthermore, the approach results in an improvement of 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 chapter 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.

Chapter 4

Timely Offloading for HPC Output Data

Modern High Performance Computing (HPC) centers are charged with supporting scientific applications that increasingly use and produce very-large datasets, e.g., analysis of neutron scattering data and deep-space observations. Of special importance are application result datasets or checkpoint snapshots from long-running simulations, which are required to be offloaded to end-user locations, where they can be analyzed for further scientific insights. For example, the Department of Energy's (DOE) Jaguar supercomputer at Oak Ridge National Laboratory (ORNL) is generating terabytes of data from user jobs from a wide-spectrum of science applications in Fusion, Astrophysics, Climate and Combustion. Result outputs from Fusion applications such as GTC [23] and GTS [106] can reach up to 40 TB and 50 TB, respectively, for a 100,000+ core run. In many cases, checkpoint data also doubles as result outputs that are used for inspecting job progress or for eventual aggregation into a set of visualized images (e.g., as in GTC, GTS, etc.). In fact, as the complexity and size of applications increase with the advent of petascale supercomputers, we may soon be faced with offloading a petabyte of data from a single application run. Another driving example is the TeraGrid collaboration [19], where result-data — from computations at any of the ten sites nation-wide — is required to be delivered to the end-user. These user facilities are accessed by a geographically distributed user-base with varied end-user connectivity, resource availability, and application requirements, delivering result-data to whom in a timely manner is a crucial challenge.

A common practice in HPC centers is to leave application-associated data management to the end-user, as the user is intimately aware of the application's data needs. However, due to the growing data demands of HPC applications, it is impractical to store all user data indefinitely at the center. HPC centers are aware of these constraints and enforce purge policies to manage the precious scratch space, wherein data is deleted based on a time window (ranging from a few hours to a few days) [5, 2]. As centers become crowded,

the purge policies get more stringent to provide space for incoming jobs. Unfortunately, there is no corresponding end-user service for a timely offload of data to avoid purging. As stated earlier, this is largely left to the user and is a manual process, wherein users stage out result-data using point-to-point transfer tools such as GridFTP [34], `sftp`, hsi [49], and `scp`.

Both end-user and center operations are negatively impacted because a comprehensive result-data offloading solution is not available. The output data of a supercomputing job is the result of a multi-hour — even several days' — run, and is usually stored on the center's scratch space. A delayed offload of such data results in sub-optimal use of scratch space in that the space is being used for a job that is no longer running. Furthermore, a delayed offload renders output data vulnerable to center purge policies. The loss of output data leads to wasted user time allocation, which is very precious and obtained through a rigorous peer-review process. Thus, a timely offload can help optimize both center as well as user resources.

In this chapter, we address the challenges associated with providing a data offloading service for HPC centers. We design a combination of both a staged as well as a decentralized offloading scheme for job output data. Compared to a direct transfer, our techniques have the added benefits of resilience in the face of end-resource failure and the exploitation of orthogonal bandwidth that might be available in the end-to-end data path. Our approach uses a collection of intermediate nodes that are specified and trusted by the end-user, thereby eliminating the concern of data delivery through a set of unreliable sites in a decentralized environment. Additionally, we develop a decision making component that factors in parameters such as a center's purge deadline, the user delivery schedule, and a snapshot of current network conditions between the center and the end-user, to determine the most suitable approach to offload. Further, we employ active monitoring, using the Network Weather Service (NWS) [107], to make the data offload process react to bandwidth degradation, thus ensuring that a user-specified delivery constraint or a purge deadline can be met. Moreover, we utilize erasure coding schemes to ensure that the offload is fault-tolerant. Finally, we have implemented the offloading service components, and have thoroughly evaluated it using both trace-driven simulations, as well as actual tests using the PlanetLab testbed [81].

4.1 Requirements of a Result-Data Offloading Service

Current solutions for offloading large data to end-user sites are often mired by several factors. First, a direct download from the HPC center to the end-user requires that end resources be available for the entire duration of the transfer. This can be a significant space and bandwidth commitment from both the HPC center and the end-user. For instance, the end-user resource might be unavailable when the data needs to be offloaded. This renders the result-data vulnerable to center purge policies. A desirable alternative, however, is to quickly move the data from center scratch space — perhaps to an intermediate storage location — so

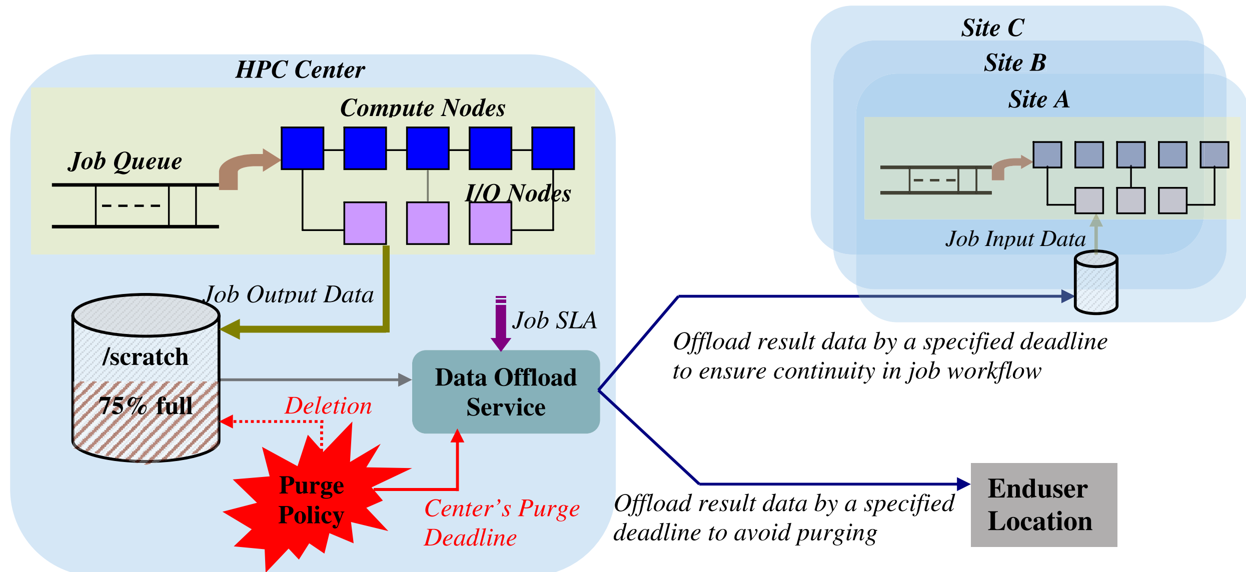


Figure 4.1: Depiction of usecases for a timely offload of result-data: (a) An expeditious offload to release center scratch space and to protect the data against a purge; (b) An end-user data delivery; and (c) Data delivery to another part of the job workflow.

that the high-end, expensive resource can be relieved. Better yet, the intermediate location can be on the data path to the end-user, so the data can be delivered from the intermediate location to the destination when the end-resource becomes available again.

Second, current data offloading schemes from HPC centers do not exploit orthogonal (residual, unused) bandwidth that might be available between two transfer end-points. Exploiting such bandwidth can help alleviate several problems endemic to data downloading, such as bandwidth volatility.

In essence, what is needed is an architecture for timely end-user data delivery that is able to reconcile both the HPC center's as well as a user's constraints amidst varying bandwidth and resource availability conditions.

4.2 Design

In the following, we first present an overview of the system architecture. Next, we discuss intermediate node selection and usage. Finally, we describe how individual components are integrated and utilized to provide the timely offloading service.

Table 4.1: Input parameters for the data offload manager.

Parameter	Description	Source
D_{purge}	Purge deadline	Center configuration
S	Job output data size	Job specification
J_{SLA}	Data delivery schedule	Center-user SLA
$< N_i, P_i,$ $BW_i >$	List, properties, and available bandwidth of intermediate nodes	Node discovery process

4.2.1 Architecture Overview

Figure 4.1 illustrates the overall offloading framework, which entails a combination of strategies both at the center and the end-user site to orchestrate the transfers. The design challenges arise from the interplay between the center's purge policy, the job submission system and the data transfers for offloading.

We design a new software component, the *Data Offload Manager*, to capture the above interactions and drive the offloading process. The *Manager* is integrated into the HPC center management software suite, and is provided with a number of critical center parameters and job descriptions to guide its operation. The *Manager* takes as input, guidelines regarding the purge deadline, D_{purge}, from the HPC center's scratch space purging system, and the job specification from the job submission system. The specifications include the output data size, S, the job's data delivery schedule as per the Service Level Agreement (SLA), J_{SLA}, and other details such as any potentially available intermediate nodes, $< N_i, P_i, BW_i >$, where P_i denotes usage properties/constraints of the node, N_i, and BW_i denotes the current snapshot of the observed NWS bandwidth between the HPC center and N_i. Table 4.1 summarizes the list of parameters used in our system.

The *Manager* uses these parameters to determine a course of action, i.e., an offload schedule, O_s, for offloading the job's output data. O_s can be either a direct center to end-user site transfer or a decentralized transfer through the intermediate nodes. The goal is to deliver the data in time, $T_{offload}$, such that:

$$T_{offload} \leq Min(D_{purge}, J_{SLA}) \tag{4.1}$$

Given the dynamic nature of the system, O_s needs to be constantly re-evaluated based on an updated $< N_i, P_i, BW_i' >$, where BW_i' is the latest snapshot of NWS measurements. Alternate routes have to be taken to meet the SLA if the re-evaluated time to offload, $T_{offload}'$, increases such that:

$$T_{offload}' > J_{SLA} \tag{4.2}$$

Parameter Specification

The offloading scheme relies on the job submission system for critical input parameters, some of which cannot be inferred from the center management software and must be specified by the end-user. For this purpose, we instrument the center's job submission system to enable end-users to provide information, e.g., delivery constraints and deadlines etc., as part of their regular PBS [32] job scripts. This has the added advantage of enabling easy integration and adaptation of our solution. The user simply submits the modified PBS script to our system on the center, which extracts the offloading-specific parameters and passes them to the offload manager.

Initiating the Offloading Process

Eager offloading has to be started to coincide with job completion so that output data can be expeditiously staged out. Thus, a desired functionality of the job submission system is to be able to automatically initiate a pre-specified process at job completion. Previous work [108] has focused on a similar goal, by instrumenting the job submission system to start user-specified direct data transfers, e.g., secure copy `scp` or GridFTP [34], upon job completion. This was accomplished by setting up separate queues for data and compute jobs, submitting the offload job to the data queue and specifying job dependencies such that the offload only begins after the compute job (dependency setup mechanisms are allowed by most modern resource managers). However, only simple user-specified direct data transfer commands were executed (e.g., scp or GridFTP) as part of offload in that work. In this dissertation, we use and extend this work to intimate the offload manager of the availability of a job's result dataset for decentralized offloading of the data, which can then initiate the offload. The presence of a center-wide offload manager has the advantage that it can perform global optimization, for example it can assign a higher priority to an offload that is on a tighter deadline than others.

A final piece in the data offload architecture is the utilization of a number of user-specified intermediate storage locations or nodes to which data from the center is offloaded, and from which the end-user site can then asynchronously retrieve the data. These nodes are specified by the user as part of the job submission script. By selecting resources that are closer (in bandwidth) to the center, the offload bandwidth utilization can be maximized and the chances of loosing data due to a purge reduced. The intermediate nodes also provide multiple data flow paths from the center to the submission site, faster retrieval speeds, as well as fault-tolerance in the face of failure.

4.2.2 Intermediate Nodes

In the following, we discuss the motivation, discovery, and utilization of intermediate storage locations in enabling a timely HPC data offloading process.

Motivation for Collaboration

The decentralized offload makes extensive use of intermediate nodes. We envision these to be nodes that are specified and trusted by the user. More specifically, consider the following collaboration scenarios that present a strong case for the participation of intermediate nodes in the data offloading process.

In today's HPC environment, supercomputing jobs are almost always collaborative in nature. In fact, a quick survey of jobs that are awarded compute time on the ORNL National Leadership Class Facility (NLCF) — through the DOE's INCITE [46] program — shows that these jobs involve multiple users from multiple institutions. This collaborative property is even more true in large national infrastructures such as the TeraGrid [19] that are among the key drivers for end-user data delivery. Jobs in the TeraGrid are usually from a *virtual organization (VO)*, which is a set of geographically dispersed users from different sites, coming together to solve a problem of mutual interest for a certain duration. In such cases, it is clear that many users, from different sites will be interested in the resulting job output data. Thus, there is a natural need to dispatch the result data to more than a single location.

This property of collaborative science can be exploited to perform a collaborative offload of job output data. Participating sites can come together to form an overlay of intermediate nodes that contribute space and bandwidth for the offload. We argue that there exists a *natural incentive* for the participating sites to do so. Such a definition of intermediate nodes makes them more reliable and alleviates a key concern of precious result-data being transferred through an unreliable substrate.

The *natural incentive* works well when the project is a large collaborative one. In this case, our work does not expect any well-established infrastructure. Instead, it attempts to piggyback on existing connectivity and residual bandwidth therein. Such a setup is well suited for long-running jobs where the overhead of intermediate-node setup is justified. More and more, we are observing that HPC jobs are either long-running, or that the same users run a large number of small jobs where the setup cost is amortized over the multiple runs.

Node Specification

The intermediate nodes are specified by the users as part of their job submission scripts. We provide special directives with which users can annotate their job scripts. These directives are parsed by the offload manager to extract and maintain a list of user-specified intermediate

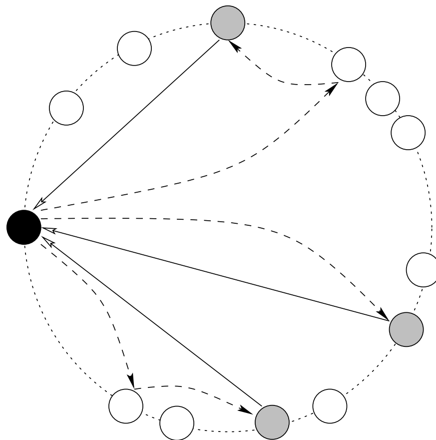


Figure 4.2: Intermediate node discovery using random p2p messages. Here, the end-user submission site (black) discovers three intermediate nodes (gray).

nodes. The explicit specification of all of the intermediate nodes that a user has access to may not be practical for large collaborations. Here, it is intended to demonstrate how a single user, even with just a handful of collaborating sites can exploit the intermediate nodes to conduct a collaborative download. In the case of large collaborations, we can imagine specifying simply the VO that the user is part of, in the job script. The data offload manager then submits the job to the scheduler. This way, the overlay of intermediate nodes becomes an integral part of the job and can be used for the delivery of the job's result data. End-users can further qualify the intermediate node specification with usage policies, which specify the available storage and the load threshold at the intermediate node. For instance, an intermediate node might be willing to participate in the collaborative offload as long as the load incurred due to the transfer is below a certain level. We will discuss this specification in more detail later in Section 4.3.

Node Discovery

The intermediate nodes are selected from among the participating sites that are interested in the data transfer. However, not all nodes are available at all times. Thus, there is a need to discover appropriate volunteer intermediate nodes (N_i's). Given the dynamic availability, and varied resource sharing policies of participants, a centralized approach would be cumbersome and impractical. Instead, we utilize the distributed and decentralized communication substrate provided by structured p2p networks [90, 98] to locate N_i's in a dynamic environment.

We use a p2p overlay (Pastry [90]) to arrange sites that intend to participate in the collaborative offload (Figure 4.2). Use of the overlay provides reliable communication with other participants in the network. The participating sites, N_i's, use the overlay to advertise their

availability to other nodes in the overlay using random broadcast. Nodes that receive these messages build local information about available nodes for offload. A given node can use its own policies and information about a remote node's capacity to make a decision regarding whether to use the remote node for the offload.

Finally, before submitting a job to the HPC center, the submission site, N_s, interacts with the center to sort the N_i's with increasing latency from the center, while at the same time with decreasing latency from N_s. A greedy approach is sufficient here, as the dynamic nature of the system takes away any advantage of trying to further optimize such ordering before the actual offload process starts. The sorted set of nodes is provided to the center to utilize as the intermediate nodes, and becomes an integral part of the job's workflow.

While not part of our current implementation, the scalable p2p discovery can also exploit the resource discovery or selection mechanisms of a VO to identify intermediate nodes within a large collaboration. VOs typically use a Monitoring and Discovery System (MDS) that maintains a list of available resources that are willing to accept jobs. In our case, this infrastructure will need to be extended to accommodate storage resources willing to donate allocations and run our service.

Landmark Nodes

The reliance of our design on intermediate nodes exposes the offload system to possible failures due to lack of sufficient N_i's. For instance, the submission site may not have access to any (or sufficient enough) intermediate nodes on the path to the HPC center. This could be either due to the lack of many participating sites in the job or due to the volatility of the intermediate nodes. To avoid such a scenario, we utilize a number of geographically distributed Landmark nodes that are always available and can serve as intermediate nodes in case enough p2p-nodes are not available. The Landmark nodes can be other HPC centers, or nodes along national links such as, Internet2 [9] Lambda Rail [12], REDDNET [6] or the TeraGrid [19] to which many end-users may be connected and have access to. The location and number of the Landmarks is determined through out-of-band agreements with the HPC center. An example application for this usecase is CERN's LHC [51] experiment, which is proposing to use national and regional sites as Tier 1 and Tier 2 data distribution centers to disseminate the experimental data from Tier 0 at CERN. Individual users can download data from these tier sites depending on geographic proximity.

4.2.3 The Data Offloading Process

The offloading process is initiated at the completion of a job as follows. First, the center chooses a number of nodes from the set of N_i's ordered by available bandwidth. The exact number of nodes used for this purpose, i.e., the fan-out, is chosen to achieve maximum (pre-specified) out-bound center bandwidth utilization, or to meet previously agreed-upon

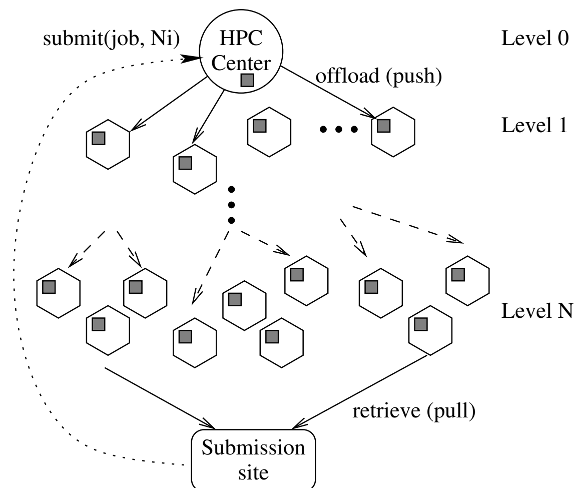


Figure 4.3: The data flow path from the HPC center to the submission site. The intermediate nodes are represented by hexagons. The participants also run an instance of the NWS (gray square) for bandwidth monitoring.

offload deadlines. These chosen N_i's serve as the Level-1 intermediate nodes. Note that the selected fan-out is not static, and can vary depending on the transfer speeds achieved. Second, the result-data is split into chunks and parallel transfer of the chunks to Level-1 nodes is initiated. Since the Level-1 nodes are much closer to the center than the submission site, the offload time is expected to be much smaller than a direct transfer to the submission site. If not, the manager would have opted for a direct transfer schedule to the end-user site, and not the decentralized offload. This has the desired effects of both releasing the precious scratch space occupied at the center and protecting the data from the purge. Third, Level-1 intermediate nodes may also further transfer data to the Level-2 intermediate nodes (once again chosen from N_i's), and so on. Consequently, data flows towards N_s, though it is not pushed to N_s. Finally, N_s can asynchronously retrieve the data from the Level-N nodes. Decoupling N_s from the data push path allows the center to offload the data at peak (pre-specified) out-bound bandwidth without worrying about the availability (and connection speed) of N_s, while enabling N_s to pull (retrieve) data from N_i's as necessary. The key steps in the offload process are illustrated in Figure 4.3.

The Push of data from one level to another (e.g., Level-1 to Level-2) is similar to the initial offload process, and is decentralized. Similar to the center, Level-i nodes may want to achieve a predetermined out-bound bandwidth, or may simply be configured to offload the data they have to a configurable number of Level-$(i+1)$ nodes for replication purposes. Either option results in choosing the fastest nodes to complete the Push operation.

The use of intermediate nodes in our system provides multiple data-flow paths from the center to the submission site N_s, leading to several alternative options for data delivery. For instance, data may be replicated across different N_i's during the transfer from one level to

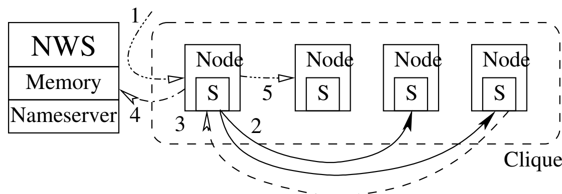


Figure 4.4: Bandwidth monitoring using the NWS. 'S' indicates our software.

another. This will allow N_s to pull data from a number of locations, thus providing fault tolerance against node failure, as well as better utilization of the available in-bandwidth at N_s. The schedule can also be used to simultaneously deliver data to multiple interested sites.

Providing Service Guarantees

The submission site and the HPC center have SLAs regarding how quickly data can be offloaded from the center. Similar to the intermediate node specification, the SLAs are also specified in a job script.

Given the dynamically changing bandwidths between participants, a fixed or statically chosen fan-out is insufficient. Therefore, we utilize a bandwidth monitoring-based scheme to dynamically adjust the fan-out and ensure meeting the SLA. For this purpose, we employ the Network Weather Service (NWS) [107] to monitor and estimate the available bandwidth between participating nodes. As seen in Figure 4.4, each participating node joins a "clique", which is a group of sensors that measure bandwidth. A token is passed around (Step 1), which serves as an indication to a node to probe (Step 2) other nodes for available bandwidth. The replies (Step 3) are recorded not only at the node, but also at a central NWS repository (Step 4). The token is then forwarded to the next node (Step 5). The clique gives the center an estimate of the bandwidth available from it to different nodes. The center uses this information to decide whether a chosen fan-out is sufficient to meet a particular SLA, or needs to be increased. If needed, additional nodes from the set of N_i's can be chosen to increase the fan-out and meet the SLA. Nodes at Level-i utilize a similar approach to determine the fan-out for Level-$i+1$. At each level, a decision making component re-evaluates the time to offload as mentioned earlier. In case the number of available N_i's are insufficient for meeting the SLA, the submission site is informed, which in turn can either provide more intermediate nodes or accept the best effort from the HPC center.

Fault Tolerance through Erasure Coding

As stated earlier, pieces of the result-data can be replicated across many participating intermediate nodes, facilitating retrieval from any subset of the nodes. In addition to this, we apply erasure coding [65, 85] to the data to improve the reliability of the transfer, while

minimizing the amount of transferred data. The computational cost of erasure coding can be paid by the Level-1 intermediate nodes if coding at the HPC center (which will be part of the job's time allocation) is an issue.

Discussion

Recent studies have shown the high rate of storage system failures [92, 82, 95] and the complexity of ensuring reliability in large-scale installations [31, 53, 87] such as the HPC scratch space. Improving reliability in such fixed installations entails going through a rigorous and time-consuming acquisition process mired with delays. In contrast, the collective use of less-reliable individual intermediate nodes can provide a solution that can be arbitrarily grown to accommodate any desired level of reliability. Thus, we argue that although individual intermediate nodes may be more prone to errors compared to single disk in an HPC center, as a system our approach is able to provide better reliability due to its flexibility. Plus, this reliability comes for free as we use resources volunteered by collaborators, which would otherwise not be used [36].

4.2.4 Design Summary

By way of eagerly offloading result-data from the center, our system avoids data loss due to the center's purge policies. This in turn allows the center to free-up precious scratch space for in-coming jobs and their data, thereby improving its serviceability. By staging the data to a network of intermediate nodes, en-route to the destination, we ensure that the offload will not fail due to end-user resource unavailability. The result-data can be pulled from the intermediate nodes as and when the end-user resource becomes available. Finally, our design provides an integrated data management solution for the HPC center, rather than leaving it up to the users, thus allowing them to focus on their applications and not bogged down by unnecessary system-level details.

4.3 Implementation

The implementation of the offloading framework comprises of about 3000 lines of C code, with the p2p substrate built using FreePastry [48] in Java. Figure 4.5 shows the architecture of the software that runs on all the participating nodes. The software also runs on the HPC center as an *Offloading Service*. The list of N_i's and the SLA are provided through the job submission script. The role of various components is as follows. The *Node Manager* is responsible for maintaining N_i's. The *SLA Compliance* module uses bandwidth predictions provided by NWS [107] (through the *NWS Query* module) to guide the offload process in meeting that SLAs. The *Erasure coding* module transforms the data to be sent out

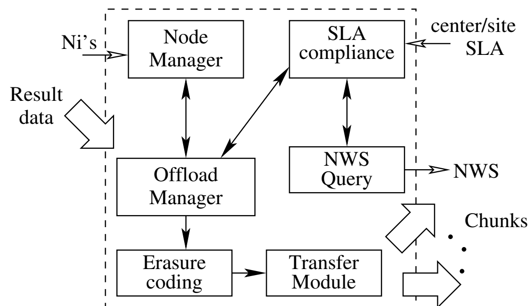


Figure 4.5: The per-node system components and their interactions.

into error-coded chunks, and the *Transfer Module* is charged with pushing out the encoded chunks to the next-level intermediate nodes. Finally, at the heart of the system is the *Offload Manager* that integrates all the modules and uses them to select different offload schedules and to enable the transfers. The erasure code that we have used is Reed-Solomon [84] in 4:5 coding configuration, i.e., four input chunks are coded to produce five output chunks, with a redundancy of 25%. The chunk-size is a tunable parameter which can be set based on the size of the datasets being offloaded.

4.3.1 Integration with Job Submission System

HPC centers utilize job management systems, e.g., batch job queuing using PBS [32], to ensure proper operation. Typically, the job submission system constitutes a user job script and a resource manager at the supercomputer center that schedules the jobs based on a queuing system. Thus, the natural place to specify user-defined intermediate nodes and deadlines is the existing job submission scripts.

To this end, we have instrumented the PBS [32] job submission system that is prevalent in HPC centers to enable specification of user-defined intermediate nodes and deadlines. We have devised a way for specifying intermediate nodes and delivery deadlines as annotations within a standard PBS script. These annotations are specified as directives, much like other

Table 4.2: New script directives used for offloading.

Directive	Parameters	Description
`Stageout`	Output dataset, destination site	Specifies offloading dataset and the target destination site
`InterNode`	IP address, bandwidth snapshot, availability, capacity	Specifies an intermediate site location
`Deadline`	Time	Specifies the deadline for the offload to complete

```
#PBS -N myjob
#PBS -l nodes=128, wtime=12:00

mpirun -np 128 ~/MyComputation

#Stageout file:///home/scratch/user/output1 file:///home/scratch/user/output1
#Stageout file:///home/scratch/user/output2 file:///home/scratch/user/output2

#InterNode node1.S1:49665:50GB
...
#InterNode nodeN.SN:49665:30GB

#Deadline 12/14/2010:12:00
```

Figure 4.6: An instrumented PBS script containing offload specific directives.

PBS directives (e.g., `#PBS`). The intermediate nodes can be further qualified with policy specifications that capture usage constraints. These constraints include the amount of space available for offload on a node, and the node's availability. More fine grained policies can be easily added.

Table 4.2 presents the directives that we have introduced to support the offloading process. Figure 4.6 shows an instrumented PBS script with these directives, wherein a user specifies the output data and the final destination, the use of intermediate nodes with their space constraints, a port number where our transfer protocol is listening, and a delivery deadline.

To handle the instrumented job script, we have implemented a parser that runs on the HPC center. When an annotated PBS script is submitted for execution to the job scheduler at the HPC center, it is intercepted by our parser that filters out directives specific to data offloading, and passes those details to the *Offloading Service* for data delivery. The remaining PBS script is then handed over to the PBS queue for standard processing. As discussed above, the *Offloading Service* is aware of the center's purge deadline and attempts to reconcile that with the user's delivery deadline and the intermediate/landmark nodes to achieve a desired data transfer schedule.

4.3.2 Integration with BitTorrent and NWS

We have designed our offloading mechanism to exploit the data dissemination abilities of BitTorrent [41] and the network monitoring facilities of NWS [107]. While both of these services are centralized in our current implementation, we note that the design provides for distributed equivalents to be built and substituted easily.

Each participating node in our system runs an NWS daemon. We have configured NWS sensors that keep track of the vital statistics of each node, as well as record bandwidth measurements between nodes. These measurements are retrieved by our *Offload Manager*

via periodic queries and used in determining appropriate offload paths that can sustain sufficient bandwidth to meet specified SLAs. The *Offload Manager* also employs the data from NWS to select additional peer nodes in case an SLA cannot be met.

The decision to add additional nodes to the offload path is driven by several factors: user-center delivery and purge deadlines, storage capacity of nodes (specified via the PBS script), and the available bandwidth.

Once a set of intermediate nodes is selected using NWS, we use BitTorrent's scatter-gather protocol to transfer the file from the center to the selected intermediate nodes. The offload happens as follows. The *Offload Manager* creates a meta-data "torrent" file for the subset of data to be transmitted to a set of chosen intermediate nodes. The Manager also provides BitTorrent *tracking* services so that the intermediate nodes know what data has been transmitted to which node. Once the nodes receive the torrent file, they use the metadata information along with the tracker to "download" the data subset to their local storage. The process is repeated at all the intermediate node levels. The end-host can also use appropriate torrent files to download the result-data from the intermediate nodes, thus completing the offloading process. Finally, issues that could arise due to the use of multiple data sources are simplified by using BitTorrent. For example, if two Level-1 nodes decide to send the same dataset to a Level-2 node, BitTorrent will automatically utilize both copies of the data at the Level-1 nodes to quickly complete the transfer.

4.3.3 Deployment

We briefly highlight some deployment issues pertaining to the design details illustrated above. Earlier, we discussed how, given a set of intermediate nodes, our approach can discover a subset of them and compose them in a scalable fashion for a collaborative data delivery. However collaborative the nature of scientific discovery, in day-to-day supercomputing environments, resource sharing often boils down to agreements between compute clusters, storage resources, and networks. The Grid community has spent a significant amount of time and effort in enabling these policies and collaborations, and we can leverage much from it. Instead, in this brief deployment discussion we put forth the concept of a *Storage Service*, a piece of software that an intermediate node can run to participate in our infrastructure. The storage service is essentially the building block for constructing our overlay and involves an intermediate node allocating a certain amount of storage (exposed at a mount point), advertising the protocols available for data transfers, and installing and running the software necessary for scalable discovery. The service also runs our BitTorrent-based data servers and clients as well as an NWS daemon. A node can choose not to run our data movement tool and instead opt for an existing transfer tool. Our data delivery mechanism will need to factor this, in addition to the advertised available storage, into the decision making. In the future, we can envision a catalog of such storage servers being maintained at a well-known location in the case where the user is part of a collaborative science team. In those cases, users need not specify the intermediate nodes in their job script, as that can quickly become cumbersome.

4.4 Simulating the Offloading Process

The interplay between the different system components at an end-user site and the HPC center is complex and requires a controlled environment for in-depth analysis, which is near-impossible to do in real HPC setups. Thus, we have developed a realistic simulator for the offloading process, *simHPC*, which models both job execution and data offloading. Appendix A contains additional detailed information about the design and inner workings of *simHPC*.

4.5 Evaluation

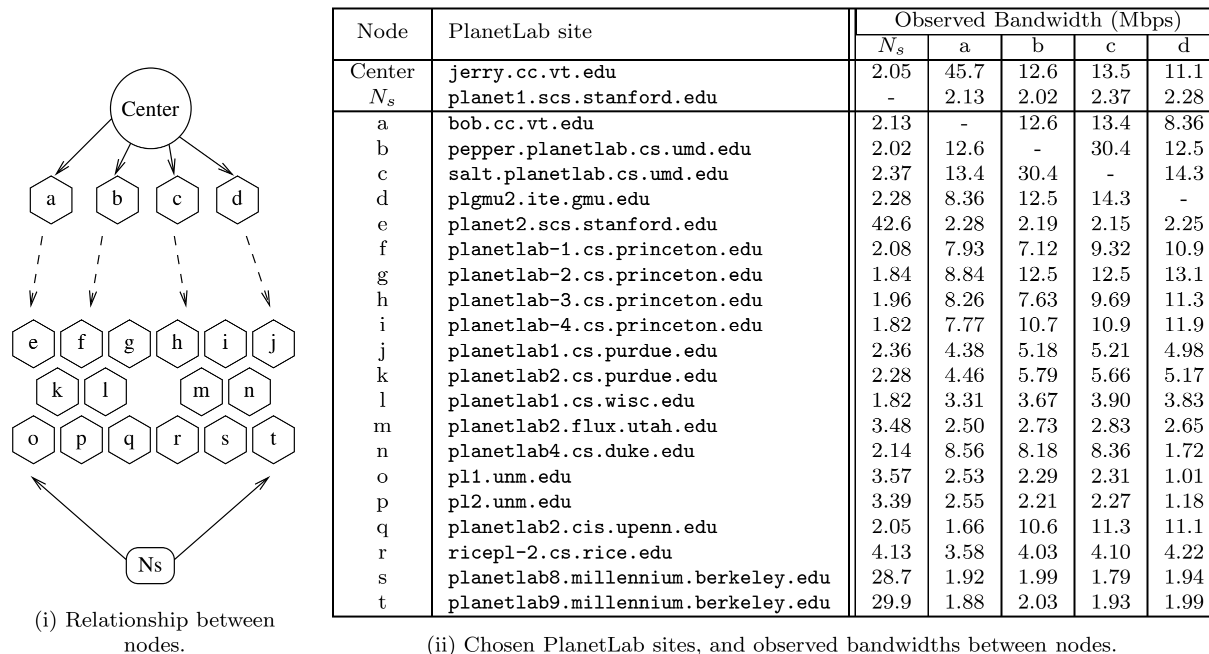
We evaluate our result-data offloading service using both the implementation of Section 4.3, and the HPC center data-subsystem simulator of Section 4.4 and Appendix A. The goal of the evaluation is to show the effectiveness of our approach to better handle data offloading.

4.5.1 Implementation Results

We emulated the dynamic behavior of the proposed data offload model using the distributed testbed facilities of PlanetLab [81]. For our experiments, we chose 22 PlanetLab sites such that the HPC center and the submission site were on opposite coasts of the US, while the rest of the nodes were geographically scattered in between. All the nodes were arranged in a tree with the HPC center as root, the number of children ranging from zero to four, and two levels of intermediate nodes. Such a tree offers multiple data flow paths from the center to the submission site and allows for testing the approach under different scenarios. Table 4.3 shows the observed average bandwidth between the center, Level-1, Level-2, and N_s nodes used in our experiments. While Figure 4.7, shows the overall experimental setup, as well as the observed pair-wise bandwidth between various nodes on the data flow path. In the following experiments the BitTorrent chunk size was set to 256 KB. Moreover, the reported numbers represent averages over a set of three runs.

Table 4.3: Inter-level bandwidth statistics.

From	To	Average BW (Mbps)	Observed Stdev (Mbps)
Center	Level-1	20.7	16.7
Center	N_s	2.05	-
Level-1	Level2	5.4	3.6
Level-1	N_s	2.2	0.2
Level-2	N_s	8.4	12.9



(i) Relationship between nodes.

Node	PlanetLab site	Observed Bandwidth (Mbps)				
		N_s	a	b	c	d
Center	`jerry.cc.vt.edu`	2.05	45.7	12.6	13.5	11.1
N_s	`planet1.scs.stanford.edu`	-	2.13	2.02	2.37	2.28
a	`bob.cc.vt.edu`	2.13	-	12.6	13.4	8.36
b	`pepper.planetlab.cs.umd.edu`	2.02	12.6	-	30.4	12.5
c	`salt.planetlab.cs.umd.edu`	2.37	13.4	30.4	-	14.3
d	`plgmu2.ite.gmu.edu`	2.28	8.36	12.5	14.3	-
e	`planet2.scs.stanford.edu`	42.6	2.28	2.19	2.15	2.25
f	`planetlab-1.cs.princeton.edu`	2.08	7.93	7.12	9.32	10.9
g	`planetlab-2.cs.princeton.edu`	1.84	8.84	12.5	12.5	13.1
h	`planetlab-3.cs.princeton.edu`	1.96	8.26	7.63	9.69	11.3
i	`planetlab-4.cs.princeton.edu`	1.82	7.77	10.7	10.9	11.9
j	`planetlab1.cs.purdue.edu`	2.36	4.38	5.18	5.21	4.98
k	`planetlab2.cs.purdue.edu`	2.28	4.46	5.79	5.66	5.17
l	`planetlab1.cs.wisc.edu`	1.82	3.31	3.67	3.90	3.83
m	`planetlab2.flux.utah.edu`	3.48	2.50	2.73	2.83	2.65
n	`planetlab4.cs.duke.edu`	2.14	8.56	8.18	8.36	1.72
o	`pl1.unm.edu`	3.57	2.53	2.29	2.31	1.01
p	`pl2.unm.edu`	3.39	2.55	2.21	2.27	1.18
q	`planetlab2.cis.upenn.edu`	2.05	1.66	10.6	11.3	11.1
r	`ricepl-2.cs.rice.edu`	4.13	3.58	4.03	4.10	4.22
s	`planetlab8.millennium.berkeley.edu`	28.7	1.92	1.99	1.79	1.94
t	`planetlab9.millennium.berkeley.edu`	29.9	1.88	2.03	1.93	1.99

(ii) Chosen PlanetLab sites, and observed bandwidths between nodes.

Figure 4.7: The experimental setup used for evaluation.

Approach Feasibility

In the first set of experiments, we determined the feasibility of our approach compared to several point-to-point direct transfer tools that are prevalent in HPC: (i) scp, a baseline secure transfer protocol; (ii) IBP [83], an advanced transfer protocol that makes storage part of the network, and allows programs to allocate and store data in the network near where they are needed; (iii) GridFTP [34], an extension to the FTP protocol, which provides authentication, parallel transfers, and allows TCP buffer size tuning for high performance; and BBCP [21], which also provides high performance through parallel transfers and TCP buffer tuning. Note that these protocols are all typically supported [24] by HPC centers such as Jaguar [18].

We used a range of file sizes from 500 MB to 5.0 GB and measured the time for each direct transfer method between the center and the submission site. For our offloading, we used a combination of BitTorrent and NWS as outlined earlier.

In Table 4.4, we compare Direct transfers with the times to offload data from the source (HPC center) to Level-1 nodes (Offload), time to forward the data from Level-1 to Level-2 (Push), and the time it takes the submission site to pull the data (Pull). Compared to the Direct transfer mechanisms, the Offload is able to release the HPC center scratch space dramatically sooner for the data sizes we considered, as shown in Table 4.4. This has a significant impact on the HPC center serviceability since the free space can now be used for new incoming jobs.

Table 4.4: Comparison of decentralized transfer times (in seconds) with different direct transfer techniques. The buffer size for IBP, GridFTP, and BBCP is set to 1 MB. The number of streams in GridFTP and BBCP is set to 8 and 16, respectively.

File Size		500 MB			2.1 GB			5.0 GB		
Decent-ralized	Offload	169			570			1339		
	Push	349			1123			2692		
	Pull	202			562			1387		
			Slowdown wrt.			Slowdown wrt.			Slowdown wrt.	
			Ofld.	Pull		Ofld.	Pull		Ofld.	Pull
Direct	scp	1443	8.5x	7.1x	5834	10.2x	10.4x	13917	10.4x	10.0x
	IBP	929	5.5x	4.6x	3660	6.4x	6.5x	8546	6.4x	6.2x
	GridFTP	359	2.1x	1.8x	1603	2.8x	2.9x	3624	2.7x	2.6x
	BBCP	273	1.6x	1.4x	995	1.7x	1.8x	2373	1.8x	1.7x

Compared to each Direct transfer mechanism, the time to pull the data to the submission site is also reduced as seen in Table 4.4. The reported pull time represents the time to transfer the file from Level-1 and Level-2 nodes to the submission site, and does not include the transfer time from the source. However, the submission site pull is asynchronous, and can start as soon as chunks begin to arrive at Level-1 nodes. We note that the overall transfer time, i.e., the time from when the source starts sending the data to when the submission site has received all the data is not a suitable metric, as our approach allows the site to be offline during the offloading process and delay starting the pull as necessary. However, the earliest time the user can get the output data is still a useful metric. In our system, the end-user can start retrieving the data as soon as the center has offloaded it to Level-1 nodes. Thus, the Offload times reported in Table 4.4 also serve as the earliest data availability metric, and as stated earlier are significantly better in our approach compared to a direct data transfer.

Dynamic Data Scheduling

In this section, we compare our approach with a regular BitTorrent-based data transfer. In this case, we use NWS bandwidth measurements to greedily provision Level-1 nodes to increase the fan-out until a maximum (predetermined) center outbound bandwidth is utilized.

Table 4.4, discussed in the previous section, shows data offloading using the bandwidth measurement-based approach. Table 4.5 shows the time taken to deliver a 5.0 GB dataset using the regular, unmodified BitTorrent protocol. Our results indicate that all three steps in our approach: Offload, Push and Pull out-perform the corresponding steps in a regular BitTorrent transfer. The Offload from the HPC center to Level-1 nodes is 52.9% faster, while the Push from Level-1 nodes to Level-2 nodes is 26.9% faster. Use of bandwidth measurements, therefore, results in reduced intermediate forwarding time. The time to pull

Table 4.5: The time to transfer a 5.0 GB file using standard BitTorrent. The equivalent phases for our scheme are shown in brackets.

Phase	Time(s)
Send one copy from center (Offload)	2844
Send to all intermediate nodes (Push)	3684
Submission site download (Pull)	1393

the file to the submission site is slightly increased by 0.4%. This is expected, as the flow paths do not affect the time it would take for the submission site to pull the file. These results show that bandwidth measurement provides a good tool for improving offload times.

Effect of Chunk Size on Offload times

In our next experiment, we varied the chunk size used by BitTorrent and observed the effects on file transfer time. The results are shown in Table 4.6. As the chunk size increases, the transfer time decreases. A chunk size of 1024 KB improves transfer speed by 6.58% when compared with the default chunk size of 256 KB. These results indicate that the transfers can benefit from larger chunk sizes.

When to Employ Staged Offload?

In the experimental setup we have adopted, the bandwidth available between the center and Level-1 nodes is greater than that between the center and N_s. Thus, in this setup, the center will always decide to perform staged offloading. In the next experiment, we modified the setup to use a node from our Level-1 nodes, i.e., a node with better connectivity to the center, as the end user site, and did not use N_s. Then, we repeated the above experiment to offload a 2.1 GB file, first, without considering direct transfer and always using the staged offload mechanisms, and second, with the ability to choose between direct and staged offload depending on the ability to meet a SLA deadline. We observed that for the first case, the time to offload and pull the data was 610 seconds and 400 seconds, respectively. In contrast, for the second case the direct transfer completed in 380 seconds, an improvement of 37.7% in offload times. This result coupled with the earlier experiments stress the need for the offload mechanisms to dynamically adjust to the variations in the system behavior and to not be hard-wired to simply always do a staged offload or a direct transfer.

Table 4.6: Relative improvement in file transfer times using BitTorrent under varying chunk sizes, compared to the default chunk size of 256 KB.

Chunk Size	128 KB	256 KB	512 KB	1024 KB
Time saved (%)	-2.14	0	5.46	6.58

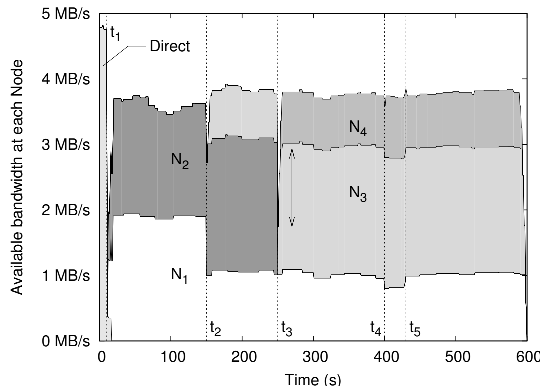


Figure 4.8: Utilized out-bound bandwidth at the center, as the system adjusts to failures and meets the 600s deadline for offloading. The labeled regions represent utilized bandwidth to individual nodes.

Enforcing SLA

In the next experiment, we study the effectiveness of the proposed approach in enforcing SLAs. We assume that the submission site and the HPC center have agreed on an SLA to offload the 2.1 GB file to four Level-1 nodes (N_1 to N_4) or a direct transfer in 600s. Initially, we choose a site that supports a large bandwidth between the center and the site. Thus, our algorithm starts off by doing a direct transfer. However, at time $t_1 = 10s$, we limit the inbound bandwidth of the site to 1/10 of its value. Soon after this happens, our system realizes that the SLA cannot be met with a direct transfer and switches to a staged offload. Once an offload schedule is chosen, we utilize bandwidth provided by the NWS to estimate the time E_t it would take to offload the remaining chunks of the file. If E_t turns out to be longer than necessary to meet the SLA, the fan-out is increased. The process is repeated every time the available bandwidth predictions change. To force dynamic scheduling to come into play, we artificially introduced two bandwidth-changing events during the offload: at time $t_2 = 150s$, we limited the available bandwidth to N_1 to about 1 MB/s; and at time $t_3 = 250s$, we failed N_2. Figure 4.8 shows the sum of the utilized bandwidths between the center and each of the four Level-1 nodes reported every second. Initially, only N_1 and N_2 are used. Soon after t_2, the drop in N_1's bandwidth is detected causing an increase in E_t. The system reacts by increasing the fan-out to use N_3, so that E_t remains under the 600s deadline. Note that between t_2 and t_3, the maximum available bandwidth of N_3 was not needed to meet the SLA and was not utilized. However, when N_2 failed at t_3, the system first uses N_3's maximum bandwidth as observed as a spike (indicated by the arrow) in N_3's curve following t_3. However, this increase is not sufficient to compensate for the loss of N_2, hence, the fan-out is adjusted to also use N_4. Also note that between t_4 and t_5, the available bandwidth for N_1 is reduced significantly enough to cause the system to utilize a higher bandwidth to N_4 so that the overall total bandwidth is maintained to meet the SLA. Once N_1's bandwidth returns to normal, our greedy algorithm once again increases the use of N_1's

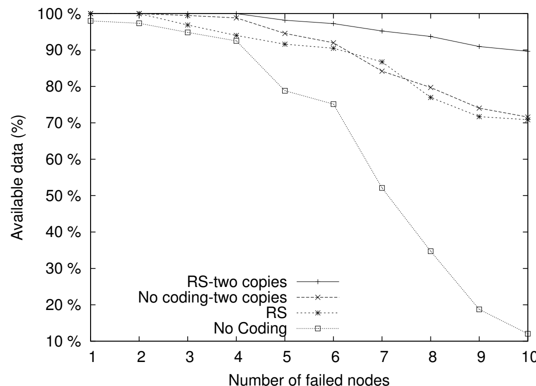
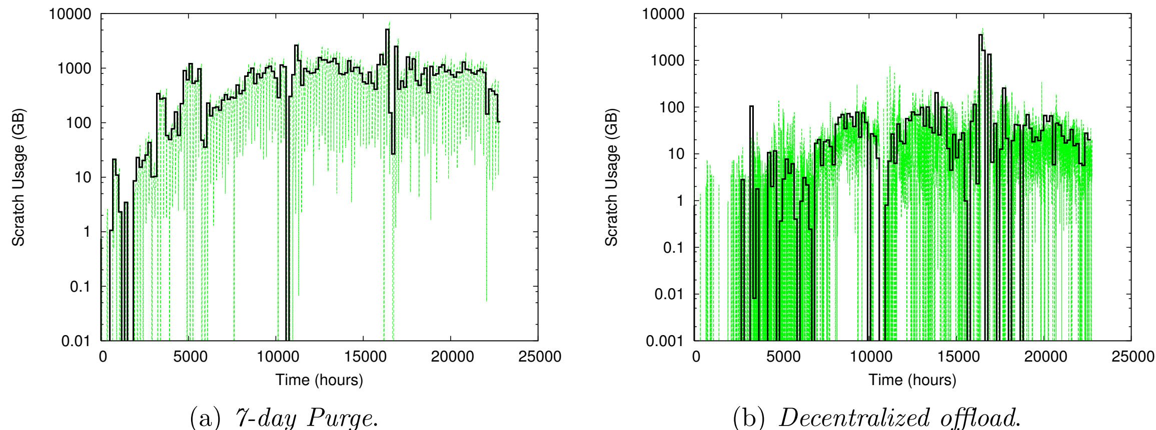


Figure 4.9: Available data under different error coding schemes, as intermediate nodes fail.

bandwidth and reduces the use of N_4's bandwidth. The two spikes at t_4 and t_5 capture the system response time to these events. Finally, as observed from the figure, the system is able to transfer the file within the specified SLA by dynamically adjusting the fan-out.

Data Availability

In this experiment, we measured the effect of Error Coding on fault tolerance. For this purpose we randomly failed several intermediate nodes during the course of the transfer and determined what portions of the file become unavailable. The experiment was repeated with an increasing number of failed nodes, up to 10 (50%). Figure 4.9 shows the average results over three runs for four scenarios: with no error coding, using 4:5 Reed-Solomon [84] coding (RS), and using replication to create two copies under both no error coding and RS. As expected, using neither error coding nor replication causes data to become unavailable even with a single failure, with up to 87.9% of data being unavailable with 10 failed nodes. Use of error coding or replication allows the file to be transferred successfully even when multiple nodes on the path from the center to the client fail. Note that both RS-single copy and replication are able to provide 100% availability with up to two (10%) node failures. This is promising as our RS code has only 25% redundancy to that of 100% of replication. However, with additional node failures simple replication is able to provide better availability than RS. Creating two copies of the data under RS further improves data availability: 100% availability when 25% of the intermediate nodes have failed, 89.7% availability with the extreme case of 50% of failed intermediate nodes. Hence, error coding at the center along with replication through multiple data-flow paths can provide excellent fault-tolerance behavior for the offloading process.



(a) *7-day Purge.* (b) *Decentralized offload.*

Figure 4.10: Overall scratch utilization over the duration of the traces under different approaches. The solid line shows the average utilization measured per hour.

4.5.2 Simulation Results

In the next set of experiments, we utilized our simulator (Section 4.4 and Appendix A) to study in detail the impact of our approach on overall scratch utilization, and towards mitigating the role of failures in job scheduling delays.

Center Log Statistics

The simulator is driven by job-statistics logs collected over a period of three-years (2004-2007) on the Jaguar [18] supercomputer. Table 4.7 shows some relevant characteristics of the logs. Also, note the large variance in both the duration of the jobs (from a few seconds to over a day) and the amount of data they access (from a few MBs to several TBs), implying that even a small amount of scratch savings for larger jobs can enable accommodating a large number of smaller jobs, consequently increasing the center's job throughput. For this study, we assume that the job input and output data sizes are capped to the total aggregate memory usage of the job. For example, if the job used 1,000 compute cores and 2GB of memory per core, we assume its output data size to be 2000GB. This is a very reasonable assumption given that many data-intensive applications' checkpoint or restart output datasets cannot be larger than their total memory usage. In the absence of per job output data size information

Table 4.7: Statistics about the job logs used by *simHPC*.

Duration	22753 Hrs
Number of jobs	80025
Job execution time	1 s to 120892 s, average 5849 s
Input data size	2.28 MB to 7481 GB, average 65.3 GB

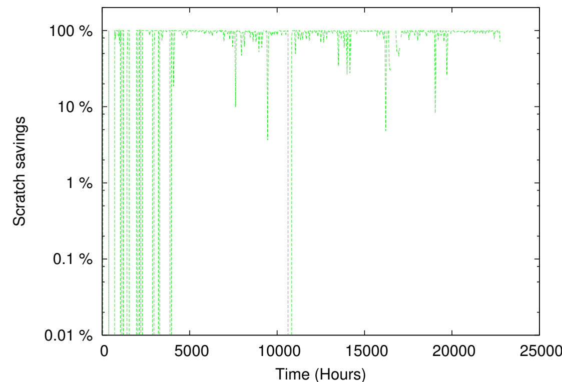


Figure 4.11: The scratch savings achieved by using a decentralized offload compared to a standard 7 day purge.

in the logs, we consider such an estimate to be a realistic approximation, capturing current usage trends. The output data itself can run in the hundreds of GBs and TBs for leadership simulations on Jaguar. For example, Fusion applications such as GTC, GTS and XGC1 produce 44 TB, 50 TB and 300 GB, respectively, of output data from runs on 100,000+ cores. Given that outputs themselves can be quite large, we did not include the effect of intermediate checkpoint snapshot data on scratch utilization.

Impact on Scratch Space Utilization

In the first set of experiments, we quantify the impact of our timely offloading approach on scratch space usage. We play the logs in our simulator and determine the amount of scratch used both under a 7-day purge policy and decentralized offloading. For this test, we assume that the scratch is empty at the beginning. Only output data is considered, and a data item is only purged if its associated job has completed. Figure 4.10 shows the scratch space usage under the studied approaches, measured every 10 minutes. Observe that the scratch utilization under decentralized offloading is (as much as an order of magnitude) lower than that under a 7-day purge.

To further illustrate the reduced scratch usage, Figure 4.11 shows the average per hour savings achieved by the decentralized offloading approach. Here, we observe that on average across the entire log, decentralized offloading uses 88.2% less scratch per unit of time (e.g. 882 GB/Hr on average per Terabyte of storage) compared to a simple 7-day purge. Thus, our approach is a promising way for conserving precious scratch resources.

Table 4.8: Job delays under the different offloading approaches.

Offload Type	Jobs Delayed			Maximum Delay (Hrs)			Average Delay (Hrs)		
	5.0 TB	2.5 TB	1.0 TB	5.0 TB	2.5 TB	1.0 TB	5.0 TB	2.5 TB	1.0 TB
7-day Purge	2541	11027	71253	59.0	351.3	7347.2	37.2	109.3	3446.0
Decent. Offloading (DO)	0	1010	2893	0.0	38.6	92.0	0.0	24.0	47.4
DO + Encoding	0	1226	3013	0.0	58.3	113.7	0.0	42.6	57.7
DO + 2 copies	114	1874	3409	2.5	82.0	142.2	1.8	54.2	77.0
DO + Encoding + 2 copies	197	1739	3207	7.6	85.8	142.0	5.4	59.2	80.8

Impact on Job Scheduling & Center Serviceability

In the next experiment, we limit the available scratch space, and study how job scheduling will be affected under a simple 7-day purge and our decentralized offloading with several redundancy improving techniques, namely, erasure coding, two data copies, and two-copies plus erasure coding. To analyze the impact of completed jobs' data offloading on the scheduling of new in-coming jobs, we measure the delay that might be incurred in starting the new jobs. New jobs will be delayed if their input data cannot be staged into the scratch space due to a lack of sufficient space, resulting from the scratch not having been cleared of result output data from the previously completed jobs. Table 4.8 shows the results in terms of the number of jobs delayed, the maximum observed delay, as well as the average delay. Compared to a simple 7-day purge, decentralized offloading can significantly reduce the delays: 78.1%, and 98.6% for 2.5 TB and 1 TB scratch size, respectively, while 5 TB under decentralized offloading experiences no delays. Moreover, introducing redundancy improving techniques also introduce delays, however, such delays are nominal compared to the 7-day purge. For instance, the average delay under decentralized offloading with both erasure coding and two-copies is 85.4%, 45.7%, and 97.7% less than that under 7-day purge for a scratch size of 5 TB, 2.5 TB, and 1 TB, respectively.

Next, we calculate the impact of observed delays in job scheduling using a new metric, *Expanded Usage Factor* (EUF), which we define as the ratio *(execution_time + data_wait_time) /execution_time*, where the *data_wait_time* is the time the output data has to wait on the scratch space after job completion before being offloaded. Our EUF metric is inspired by the widely-used *expansion factor* [19, 18], which is often used to quantify job delays in HPC centers. Expansion factor is defined as the ratio *(wall_time + wait_time)/wall_time* averaged over all jobs (the closer to 1, the better). Similarly, EUF indicates the extra, avoidable time for which a dataset occupies the scratch space, and the closer its value is to one, the better. Thus, EUF also provides a valuable measure of the HPC center serviceability in terms of precious scratch space consumption. Figure 4.12 shows the average EUF for the HPC center, for a duration of three years and 80,025 jobs, under different scratch sizes. Once again, the decentralized offloading (even with erasure coding and two data copies) behaves superior

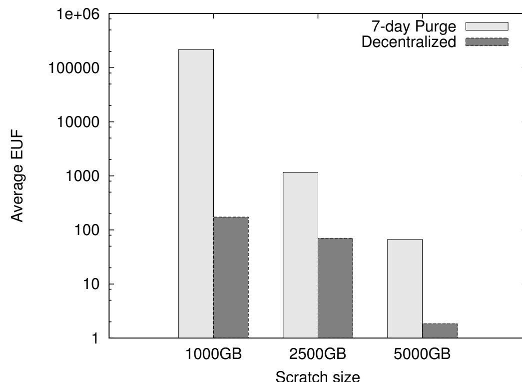


Figure 4.12: Average Expanded Usage Factor for the HPC center for a duration of three years and 80,025 jobs, under a 7-day purge and decentralized offloading with erasure coding and two-copies for varying scratch sizes.

to a 7-day purge with an average EUF reduction of 99%, 94% and 97%, with an available scratch size of 1 TB, 2.5 TB and 5 TB, respectively. Observe that even when the scratch space is 5 TB, i.e., well beyond the job-trace footprints, the decentralized approach provides a much better EUF.

Impact of Failures

Next, we measure how first-level intermediate node failures during the decentralized offloading process impact job scheduling. The total number of intermediate nodes used in this study is 25, and we assume that 10% to 50% of these nodes fail randomly during the course of an offload. Table 4.9 shows the corresponding delays under the studied scenarios. It is observed that compared to the case of no failures, intermediate-node failures can significantly delay job scheduling. However, as the failures increase from 10% to 50%, the average delay remains under 68.3%, 24.5%, and 25.3% for 5 TB, 2.5 TB, and 1 TB scratch size, respectively.

Next, Figure 4.13 shows how first-level intermediate node failures affect the EUF. A large number of first-level node failures may result in retransmissions to ensure data is not lost, which in-turn may cause the offload process to take longer. It is observed that with a

Table 4.9: Observed job delays under decentralized offloading with erasure coding and two-copies, when 10, 25, or 50 percent of the first-level intermediate nodes have failed.

Percent Failed	Number of Jobs Delayed			Maximum Delay (Hrs)			Average Delay (Hrs)		
	5.0 TB	2.5 TB	1.0 TB	5.0 TB	2.5 TB	1.0 TB	5.0 TB	2.5 TB	1.0 TB
10 percent	1129	2613	4991	66.9	156.8	221.0	42.9	90.2	96.0
25 percent	1217	2687	5405	82.1	185.1	248.4	54.0	108.3	101.4
50 percent	1484	3553	7059	122.9	231.4	325.5	72.2	112.3	120.9

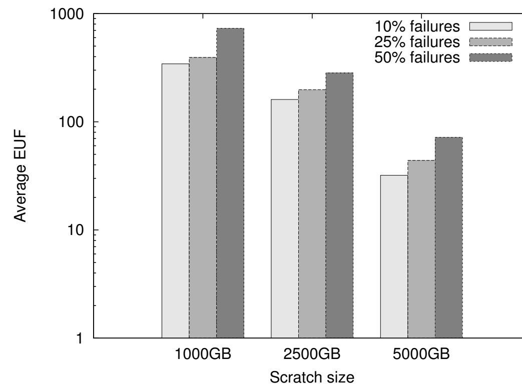


Figure 4.13: Average Expanded Usage Factor under decentralized offloading with replication and erasure coding for varying scratch sizes and different intermediate node failure rates.

constrained scratch size of 1 TB, the average EUF is increased by 98%, 127%, and 323% for 10%, 25%, and 50% of the nodes failing compared to the no failure case of Figure 4.12. Similar failure effects are observed for other scratch sizes.

In summary, the decentralized offloading approach is promising in its ability to reduce job scheduling delays, improving expansion factor, and can tolerate failures without drastically degrading overall system performance.

4.6 Chapter Summary

In this chapter, we have presented the design and implementation of a result-data offloading service for HPC centers. Offloading large data to end-user locations in a timely manner is critical to center operations, its availability and serviceability. Our approach presents a fresh look at offloading by using a set of user-specified intermediate nodes to construct a p2p network and transferring data based on bandwidth-adaptation. Our results indicate that our offloading approach can improve the rate at which the data is offloaded from the center, while allowing the submission site to pull the data as and when the site becomes available, at a much higher transfer rate because the result-data has already been staged closer. Further, offloading enables us to deliver data based on a previously agreed upon SLA, dynamically varying the fan-out as necessary. Thus, our scheme can be extremely useful to both HPC centers and users.

Chapter 5

Just-In-Time Staging of HPC Input Data

The advent of extremely powerful computing systems such as Petaflop supercomputers, and the data they can process such as very-high-resolution space observations, are pushing the envelope on dataset sizes. For instance, the Large Hadron Collider (LHC) [43] or the Spallation Neutron Source (SNS) [15] will generate petabytes of data. These large datasets are processed by a geographically dispersed user base. Therefore, result output data from High Performance Computing (HPC) simulations are not the only source that is driving dataset sizes. Input data sizes are also growing many fold [43, 15, 94, 10].

To match the I/O capabilities with the computational power in an HPC center, the required input data for a given job is almost always copied or *staged* to a fast local storage at the center — the scratch parallel file system — before the job is started. The use of scratch is strongly encouraged, as the alternative of accessing data remotely while a job is executing on the typically large number of resources creates unnecessary stalls and wastes precious allotted compute time, consequently reducing overall efficiency. Modern applications usually encompass complex analyses, which can involve staging large input data using point-to-point transfer tools such as scp, hsi [49], and GridFTP [29], from observations or experiments. Moreover, the sources of data are increasingly becoming dispersed as scientists tackle complex problems, e.g., near real-time modeling of adverse weather [26], which depend upon large swaths of data originating from distributed sensors. Thus, input data can originate from multiple sources ranging from end-user sites, remote archives (e.g., HPSS [44]), Internet repositories (e.g., NCBI [78], SDSS [94]), collaborating sites and other centers that run pieces of the job workflow.

As discussed in Chapters 1 and 3, scratch is expensive — costing millions of dollars for state-of-the-art supercomputers, e.g., Jaguar's [18] scratch has 14,000 disks, 192 object storage servers, 1300 object storage targets and 48 controller pairs — and consumes a notable fraction of the HPC center's operations budget. More importantly, scratch is meant for facilitating

currently running or soon to run jobs. This usecase precludes simple scratch management policies, e.g., quotas or charging for space usage are rarely used so that currently running jobs will not fail due to lack of space. However, from a center standpoint, sub-optimal use of scratch could impact the center's serviceability, i.e., the ability to serve more incoming jobs. That is why, even with huge scratch capacities, supercomputer administrators constantly trim usage through purge policies and weekly reminders to users to move their data from scratch. From a user standpoint, the input data is exposed to potential unavailability due to storage system failure [92, 82, 95] while it is waiting on the scratch. Consequently, when the job is started, crucial pieces of input data may be unavailable, requiring a rescheduling (a delay on the order of hours to days). What is needed is a framework that enables timely staging of large input datasets for jobs.

In this chapter, we present a JIT staging framework that attempts to have the data available at scratch, from multiple input sources, just before the job is about to run. The framework proactively brings the data to intermediate storage locations on the path from the end-user site to the HPC center. This reduces the time for copying the data to scratch, thus providing better opportunities for JIT staging. Our staging framework employs an innovative combination of high-efficiency data dissemination (BitTorrent [41]) and network monitoring (NWS [107]) to exploit orthogonal, residual bandwidth and to dynamically adapt to network volatility, respectively, to improve overall scratch utilization. Further, the overarching unique goal of our work is to reconcile scratch space consumption with volatility (both network and storage) and timely staging, which is in contrast to existing works on decentralized transfers [59, 58, 30, 57, 79]. Finally, we evaluate our staging framework using both real-world experiments as well as extensive simulations using three years worth of job logs from the ORNL Jaguar supercomputer [18].

5.1 Design

Staging the data to be coincident with job startup, i.e., Just-in-time (JIT), is challenging. First, we need to know when the user's job will commence. This has been explored extensively [97, 47], and HPC schedulers (e.g., PBS Pro [13], Moab [69]) can also provide a batch queue wait time estimate based on current and historical (jobs with a similar profile) data. However, a simple and direct use of batch queue predictions in JIT staging is not appropriate due to sudden changes in schedules. For example, an unexpected failure can cause a 10,000-core job to suddenly exit, resulting in many jobs being promoted to "ready to run" state, all too quickly.

Second, we need an estimate of how long the data staging would take from the input locations to the HPC center. We need continuous bandwidth measurements so they can be factored in to revise the route dynamically and adapt to changing network conditions. The upshot is that both the queue wait time estimates and network bandwidth estimates are volatile and "soft." Consequently, our staging solution needs to be resilient to adapt to these transient conditions.

Figure 5.1: Overview of our staging framework, and interactions between the components.

5.1.1 Architecture

In the following, we detail our framework components, and how they are integrated to realize just-in-time staging. Figure 5.1 shows the high-level system overview, and illustrates interactions between our framework components.

Intermediate Nodes

Our framework uses intermediate nodes (N_is) that can provide temporary storage for data on the path from the source to the HPC center. The intuition behind using N_is is that nodes closer to the center than the user site can support faster data transfers for staging and reduce staging times. This provides for delaying the staging to much later than when using a direct transfer, which also reduces E_w.

Given the collaborative nature of large scientific applications, these nodes can be the collaborating sites, from where other input data can also be staged, ensuring that the data are transferred through a dependable substrate. Using these nodes, the HPC center can also asynchronously retrieve data from other sources, decoupled from the user site. Intermediate nodes provide multiple data flow paths from the user site to the center, which lead to better bandwidth utilization, faster staging speeds, as well as fault-tolerance in the face of failures.

Motivation for Collaboration

In today's HPC environment, supercomputing jobs are almost always collaborative in nature. In fact, a quick survey of jobs awarded compute time on the ORNL NLCF, through the DOE's INCITE [46] program, suggests that these jobs involve multiple users from multiple institutions. This collaborative property is even more true in TeraGrid [19], where jobs are usually from a *virtual organization*, which is a set of geographically dispersed users from different sites, coming together to solve a problem of mutual interest for a certain duration. An example of this usecase is the Earth System Grid [4], where it is not uncommon for different research groups to voluntarily replicate climate model data. In such cases, it is clear that many users, from different sites will be interested in seeing the job run to completion, with as little delay as possible. This emerging property of collaborative science can be exploited to perform a collaborative staging of job input data. We therefore argue that there exists a *natural incentive* to provide resources for the JIT staging process, and that such resources are essential to the application itself and should not be construed as an "extra" component needed solely for JIT staging.

Queue Prediction as Staging Deadline

In our design, the HPC center is expected to support a batch queue prediction service (e.g., NWS batch queue prediction [7]), which the users can query before submitting their jobs to get an estimate of queue wait times. Such a service is of interest to both centers as well as users as centers get more and more crowded. Scheduling based on queue wait times is already popular in TeraGrid [19] supercomputer centers. In fact, modern resource managers (e.g., Moab [69]) are beginning to provide services that would enable users to query and obtain start times of queued jobs. The prediction service can usually provide both wait time estimates as well as the probability of a job starting by a user-specified deadline [7]. In cases where direct wait time predictions are unavailable, the user can pose a query to the service, with a deadline, and determine the likelihood of the job starting by the deadline. A 90% or higher probability can be treated as an affirmation of the user-specified deadline and can be used as the job startup time and, consequently, the staging deadline.

However, the job can potentially start earlier than this predicted deadline due to inaccuracies in the prediction or due to failure of other running jobs. Similarly, a lower probability may mean that the job may not commence by the user-specified deadline, but is only an estimate. To accommodate this, we can let the user tweak the estimate by up to a fixed factor, f, moving the deadline earlier. However, limiting the adjustment to only a factor is necessary to ensure global fairness in the staging of all jobs [74]. Consequently, the estimate is reported to the *staging manager* so it can ensure that the user-submitted deadlines are within the factor.

Algorithm 1 The timely staging algorithm.

$Job = CreateJobScript(< N_i, P_i >, D_j, BW_i)$
$T_{Predict} = GetJobStartUpPredictionFromBQP(Job)$
$T_{JobStartup} = ManagerReconcile(Job, T_{Predict}, f)$
for Each D_j **do**
  Determine X_j such that:
  $T_j = Min(DirectTransfer_j, DecentralizedTransfer_j)$
  $ScheduleTransfer(T_j)$
end for
repeat
  $BW'_i = GetNWSUpdate(BW_i)$
  $T'_{JobStartup} = GetBQPUpdate(T_{JobStartup})$
  for Each T_j **do**
    $T'_j = Recalculate(T_j, < N_i, P_i, BW'_i >)$
    if $T'_j > T'_{JobStartup}$ **then**
      Increase the $Fan - in$
    end if
  end for
until Staging Completes

Timely Staging Algorithm

Once a deadline for completing the input data staging is determined, the user submits a job script to the staging manager at the center with a description of the job and other details necessary for timely staging. The script includes attributes such as the user-adjusted job startup deadline, the set of intermediate nodes, $< N_i, P_i >$, where P_i denotes the usage properties of the intermediate node N_i, for the decentralized staging process, and the sizes and locations of the input datasets, D_j. The staging manager also takes as input the current snapshot, BW_i, of the observed NWS bandwidth between the HPC center and N_i as well as between the N_i's themselves. While we currently use bandwidth snapshots, we note that our model is independent of a specific network "distance" metric and can work with other possible metrics. This could include bandwidth, latency, as well as considerations such as out-of-band agreements.

Algorithm 1 shows the pseudo-code for the JIT staging manager. The manager reconciles the predicted job start deadline with the user-adjusted one to determine if it can allow the user's tight deadline. This reconciled deadline is denoted by $T_{JobStartup}$. Based on these parameters, the manager decides upon a data staging schedule, X_j, for each D_j, which delivers the dataset in time, $T_j = Min(DirectTransfer, DecentralizedTransfer)$. To estimate these times, the manager uses the measured available bandwidth to the user site as well as the intermediate nodes. To create a distributed schedule, the intermediate nodes are sorted based on available bandwidth and then the number of nodes to which data is sent is increased until

overall transfer times that are better than a direct transfer (if possible) can be achieved. This choice is dictated largely by the available bandwidth and storage at the intermediate nodes. When the intermediate nodes can provide a faster transfer, a decentralized transfer is scheduled. Each dataset could come from a variety of sources, including those wherein our decentralized transfer software cannot be installed. In such cases, the manager relies on *just-in-time probes* to the data source to judge if a direct transfer to the HPC center is most appropriate. Alternatively, such input data could also be transferred through the intermediate nodes by having the edge-level nodes pull the data from the source, enabling decentralized staging.

The multi-input staging should obviously also complete before job startup and should satisfy the property, $Max(T_j) \leq T_{JobStartup}$. Minimizing transfer times by choosing the intermediate nodes with the best available transfer rates helps achieve this goal. At the same time, each input staging, X_j, is also started as late as possible to reduce the duration of scratch space consumption and, consequently, the exposure window, E_w of the datasets. The exposure window for each input dataset is: $E_{wj} = T_{JobStartup} - T_j$. Then, total exposure of all input data is, $E_w = Sum(E_{wj})$. The closer E_w is to 0, the better. Thus, the ideal start time for each input dataset is the one that achieves, $T_{JobStartup} - T_j = 0$. In practice, however, a small difference is desirable to safeguard against unexpected delay. This approach factors in both timely delivery as well as scratch space usage optimization.

Re-evaluating Staging Decisions

Even after a particular course of action, e.g., decentralized transfer, is chosen, the manager periodically re-evaluates the staging (Algorithm 1) based on an updated $< N_i, P_i, BW'_i >$, where BW'_i is the latest snapshot of bandwidth measurements. If the reevaluated time to staging, T'_j, satisfies the property, $T'_j > T_{JobStartup}$, then, alternate (available) routes are taken to stage the data before job startup, enabling us to meet the staging deadline.

In addition to re-evaluating the network routes based on updated bandwidth measurements, the staging manager also has to account for batch queue status changes discussed earlier. We address this by having the manager periodically obtain new estimates $T'_{JobStartup}$ from the batch queue service. If the staging schedules reflect that $T_j > T'_{JobStartup}$, then alternate routes are evaluated to ensure timely delivery. A side effect of this is the prevention of job scheduler starvation due to inability to schedule jobs as a result of unfinished stagings.

5.1.2 Supporting Timely Staging

Once the data staging is initiated, the client chooses a number of nodes from the set of N_i's ordered by available bandwidth. Figure 5.2 shows the data flow from end-user site to the HPC center. These chosen N_i's serve as the Level-1 intermediate nodes. Note that the selected N_i's are not static, and can vary depending on the actual transfer speeds and the

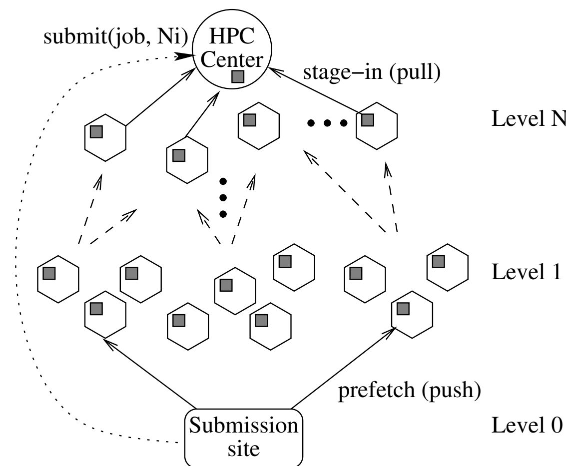


Figure 5.2: The data flow path from the client site to the HPC center. Each intermediate node (hexagon) runs NWS (gray square) for bandwidth monitoring.

impending deadline. The manager monitors the changing bandwidths periodically (using NWS) to determine if more N_i's need to be added. Next, the input data is split into chunks and parallel transfer of the chunks to Level-1 nodes is initiated. The transfer may also involve further levels of intermediate nodes (up to Level-N). The choice of the number of levels of intermediate nodes is left to the users, and does not have a direct bearing on the center to Level-N node performance that is critical for our design. The levels simply enable users to provide multiple data-flow paths to the center, and we foresee the levels to be not more than two in typical scenarios. Additionally, depending on the availability of intermediate nodes, the client can also stage the data to Level-N nodes much earlier than the deadline.

As the job startup deadline approaches, the proximity of the Level-N nodes to the center allows them to quickly move the input data to the center's scratch space. The JIT manager can vary the fan-in, i.e., the number of Level-N nodes from where to simultaneously retrieve data. The cardinality of the fan-in is chosen to stage all the necessary data before the predicted job start time (Algorithm 1). The fan-in is expanded until the deadline can be met or until no more nodes can be added. The goal is to obtain the best possible transfer time given the intermediate nodes and job deadline. Also, this design allows the Level-N nodes to stage the data at peak (pre-specified) bandwidth at the most appropriate time without worrying about the availability (and connection speed) of the submission site (Figure 5.2).

Intermediate nodes provide multiple data-flow paths as well as several alternative options for data delivery. Such multiple paths open up the possibility of a number of alternate approaches for data flow. For instance, data may be replicated across different N_i's during the transfer from one level to the other. This lets the center to pull data from a number of locations for fault tolerance.

The amount of data transferred between the intermediate nodes will vary depending on the number of nodes used and the above parameters, as well as the network conditions at the time of the transfer. However, as stressed earlier, the intermediate nodes are provided by collaborators (e.g., as in TeraGrid or ESG) that already have an interest in seeing the job succeed, and any overhead due to retransmissions between intermediate nodes can be considered as necessary to this end.

5.1.3 Discussion

Impact of storage system failures

Recent studies have shown the high rate of storage system failures [92, 82, 95] and the complexity of ensuring reliability in large-scale installations [31, 53, 87] such as the HPC scratch space. Improving reliability in such fixed installations entails going through a rigorous and time-consuming acquisition process mired with delays. In contrast, the collective use of less-reliable individual intermediate nodes can provide a solution that can be arbitrarily grown to accommodate any desired level of reliability. Thus, we argue that although individual intermediate nodes may be more prone to errors compared to individual disks in an HPC center, as a system our approach is able to provide better reliability due to its flexibility. Plus, this reliability comes for free as we use resources volunteered by collaborators, which would otherwise not be used [36].

Impact on Infrastructure Costs

We reiterate that our design does not require the explicit setup and management of landmark or intermediate nodes. Instead, it leverages and "piggybacks" on *existing* infrastructure. Several national testbeds, e.g., TeraGrid [19], REDDNET [14], etc., are already in production and can act as such nodes, without incurring any additional costs such as electricity, manpower, and management costs. Moreover, intermediate nodes use resources that are already part of the "collaborative" job. We also do not require extra provisioning of network bandwidth, rather we employ the residual bandwidth that would have otherwise gone unused. Nonetheless, extra usage, if necessary, can be construed as necessary for completing of the collaborative job, and the burden can be shared by all the collaborators, not unlike when researchers have to utilize extra resources individually to support a demanding job. Overall, our design also achieves better utilization of resources and possibly a higher system-wide efficiency.

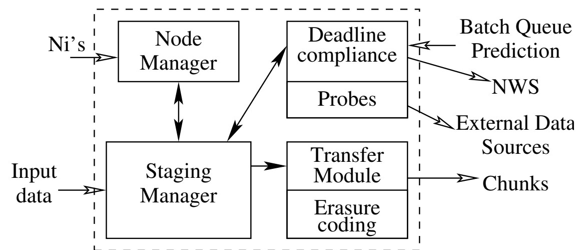


Figure 5.3: Implementation architecture for timely staging.

Alternative Data Staging Designs

There are several possible alternative solutions for the HPC staging problem, namely, adding more scratch space, streaming data directly and not using the scratch space, and moving computations closer to data. In the following, we discuss why we did not adopt these options in our design.

First, we reiterate that simply adding more scratch is not practical, as scratch is a precious commodity and provisioning more scratch means taking dollars away from buying FLOPS, and more FLOPS are how most HPC acquisition proposals are won.

Second, streaming data online and bypassing scratch to support HPC applications is not viable and sustainable (based on Top500 supercomputers). Additionally, distributed filesystems or middleware are seldom an option for extreme-scale, leadership class machines. The scratch space is a parallel file system that is made available at a mount point, to the hundreds of thousands of compute cores where the parallel job's processes run. Serving the hundreds of thousands of compute processes of a currently running job through remote I/O to a distributed file system that is geographically dispersed is a significantly expensive option, and an impractical one. Furthermore, streaming mechanisms cannot match the I/O rates required to keep such large systems busy, e.g., Jaguar's [18] scratch offers I/O rates of 256 GB/s. In fact, as pointed out earlier, HPC centers spend millions of dollars provisioning and optimizing scratch exactly to avoid this scenario.

Third, moving computation closer to data is a compelling idea, but there are numerous HPC applications, e.g., DOE supercomputer and NSF TeraGrid applications, which cannot be sustained on users' local clusters where data may be available. Our design takes all these factors into consideration for realizing a practical solution to the staging problem.

5.2 Implementation

We have implemented the JIT staging manager using about 3500 lines of C code. Figure 5.3 shows the overall architecture as well as the interactions between the manager components.

```
#PBS -N myjob
#PBS -l nodes=128, walltime=12:00

mpirun -np 128 ~/MyComputation

#Stage file://SubmissionSite:/home/user/input1 file:///home/scratch/user/input1
#Stage wget://WebRepo:/input2 file:///home/scratch/user/input2

#InterNode node1.Site1:49665:50GB
#InterNode nodeN.SiteN:49665:30GB

#JobStartDeadline 11/14/2011:12:00
```

Figure 5.4: An instrumented PBS script containing the directives for timely staging.

5.2.1 Integration with Job Submission

To facilitate easy adoption of our scheme by the community, we have integrated it with the widely-used PBS [32] job submission system. Specifically, we have instrumented the job submission scripts to let users specify intermediate nodes and deadlines. An example instrumented PBS script is shown in Figure 5.4, where the user specifies intermediate nodes and deadlines as well as details such as available storage capacities. The nodes listed in the script are just a suggestion, and the actual runtime queries these nodes directly for availability as needed.

The annotated script is submitted to the staging manager on the center, which filters out the staging-specific directives and forwards the remaining script to the standard batch queue, but with a dependency on the staging task. We extend our earlier work [73] on instrumenting the job submission system for this purpose.

5.2.2 Integration with BitTorrent and NWS

We exploit BitTorrent's [41] scatter-gather protocol for transferring data by extending the protocol to use NWS bandwidth measurements. The NWS measurements are integrated with BitTorrent to dynamically select fast locations where a particular dataset can be retrieved, and adapt to changing network behavior by adjusting fan-in to enable staging of data in time.

Since our system uses BitTorrent, the source only needs to send one copy of the data to the intermediate nodes. Once complete, if bandwidth is a consideration the source can stop "seeding", and the intermediate nodes will propagate data among themselves. However, if the source stays online after the "client offload" completes, the transfer could be quicker. Additionally, the HPC center will only need to pull one copy of the data to complete the staging process.

5.2.3 Center-wide Global Staging Considerations

Since we anticipate that all jobs, along with their staging needs, will be submitted through the staging manager, we have instrumented into the manager certain global optimizations that can be performed across all jobs. (1) All jobs that desire a staging to the Level-N, i.e., one hop away from the center, can be started immediately. Since these staging operations do not use any center resources — neither occupying scratch space nor consuming bandwidth — the data can be brought closer to the center and pulled in much faster when needed. (2) A job whose startup deadline tightens during the course of a previously initiated staging will be given higher priority if it is determined that the staging may not complete in time. For instance, this could mean providing more flows to maximize the last leg of the transfer, using more of the center's in-coming bandwidth.

5.2.4 Ensuring Data Reliability

To ensure that data is reliably staged on the center, we employ replication of data by sending out chunks to more than a single location. This is a tunable parameter in our implementation and users can specify the minimum number of replicas that should be created for a given dataset. If necessary, more space-efficient erasure codes can be used. The erasure code that we have used in our implementation is Reed-Solomon [84] in 4:5 coding configuration, i.e., four input chunks are coded to produce five output chunks, with a redundancy of 25%. The chunk-size is also a tunable parameter which can be set based on the size of the datasets being transferred.

5.2.5 Multi-Input Staging

Our implementation is capable of retrieving data from more than a single source, directly as well as incorporating it into the decentralized transfer. The data sources are provided as links in the job-submission script. If the external data source runs an instance of our software, the staging manager can simply use the NWS information to decide between direct or decentralized staging. However, if the external source does not support NWS, the staging manager uses small scale tests, e.g., a partial download from a web repository, to determine expected transfer times and make staging decisions. In this case, the goal of the staging manager is to ensure staging of all input data from all sources before the predicted job startup time.

Table 5.1: Average observed bandwidth between PlanetLab nodes during experimentation. All numbers are in Mb/s.

	Center	Client	Level-1	Level-2
Center	-	3.82	-	10.9
Client	3.07	-	5.22	-
Level-1	-	3.86	-	4.22
Level-2	9.47	-	5.66	-

5.3 Simulating HPC Data Staging Process

To systematically study the staging process in detail, we have developed a realistic simulator, *simHPC*, which models many aspects of the HPC environment including both job execution and data staging. *simHPC* also provides accounting and statistics about the staging process, such as the scratch space used and the data read, as well as other vital statistics. *simHPC*'s features and capabilities are throughly described in Appendix A.

5.4 Evaluation

In this section, we present an evaluation of our timely data staging using: (i) the implementation discussed in Section 5.2, running on the PlanetLab testbed [81]; and (ii) the HPC center data-subsystem simulator of Section 5.3 and Appendix A, which is driven by three-year job logs from the Jaguar [18] supercomputer. We also compare our JIT staging to commonly-used direct transfer techniques for staging input data in HPC centers.

5.4.1 Implementation Results

First, we use the PlanetLab [81] testbed to study the effectiveness of our decentralized staging in a true distributed environment. We chose 20 PlanetLab nodes arranged in a tree-structure: one as the client site and root of the tree, one as the HPC center, 10 and 8 Level-1 and Level-2 nodes (Figure 5.2), respectively. Table 5.1 shows the average bandwidth observed between the nodes during the course of our experiments. Our results represent averages over a set of three runs.

Decentralized JIT Staging vs. Direct Transfer

In this experiment, we compare our decentralized JIT staging to several point-to-point direct transfer tools that are prevalent in HPC: (i) scp, a baseline secure transfer protocol; (ii) IBP [83], an advanced transfer protocol that makes storage part of the network, and

Table 5.2: Comparison of decentralized transfer times with different direct transfer techniques. The buffer size for IBP, GridFTP, and BBCP is set to 1 MB. The number of streams in GridFTP and BBCP is set to 8 and 16, respectively.

Step	Transfer time (s)		
	1 GB	2 GB	5 GB
`scp`	1730	3588	8137
IBP	911	1908	4561
GridFTP	965	1841	4404
BBCP	922	1875	4745
Client Offload	703	1264	4082
Center Pull	155	337	731

allows programs to allocate and store data in the network near where they are needed; (iii) GridFTP [34], an extension to the FTP protocol, which provides authentication, parallel transfers, and allows TCP buffer size tuning for high performance; and BBCP [21], which also provides high performance through parallel transfers and TCP buffer tuning. Note that these protocols are all typically supported [24] by HPC centers such as Jaguar [18].

For this experiment, we used a range of file sizes from 1 GB to 5 GB (limited by PlanetLab policies), and measured the time for each direct transfer method between the center and the submission site. For JIT staging, we used a combination of BitTorrent and NWS as outlined earlier. Table 5.2 shows the times for the direct data transfer techniques from client to HPC center (`scp`, IBP, GridFTP, BBCP), from client to Level-1 nodes (Client Offload), and from Level-2 to the center (Center Pull). Compared to a direct transfer, decentralized staging can potentially reduce the last-hop (equivalent to Center Pull) transfer times by 91.0% and 91.0% for `scp` and by 83.9% and 83.4% for GridFTP, for 1 GB and 5 GB data sizes, respectively. This implies that the decentralized staging can potentially delay copying of data to scratch space by a factor of 11.0 for `scp` and 5.9 for GridFTP on average across the studied file sizes, and still get the data to the center in time for the job to start. Thus, it reduces the time the scratch space has to hold the data, consequently, reducing the exposure window (E_w), and improving center serviceability.

The reported Center Pull time represents the time to transfer the file from Level-1 and Level-2 nodes to the center, and does not include the transfer time from the source. However, the Center Pull is asynchronous, and can start as soon as chunks begin to arrive at Level-2 nodes. We note that the overall transfer time, i.e., the time from when the source starts sending the data to when the center has received all the data is not a suitable metric, as our approach allows the center to delay starting the pull as necessary. However, the earliest time the center can get the input data is still a useful metric. In our system, the center can start retrieving the data as soon as the client has offloaded it to Level-1 nodes. Thus, the Client Offload times reported in Table 5.2 also serve as the earliest data availability metric, and as stated earlier, are significantly better in our approach compared to a direct data transfer.

Table 5.3: The time to transfer a 2 GB file using standard BitTorrent. The equivalent phases for our scheme are shown in brackets.

Phase	Time (s)
Send to intermediate nodes (Client Offload)	1428
Download at HPC center (Center Pull)	362

Effect of Using NWS Measurements

Next, we compare our NWS-based monitored transfer approach with a standard BitTorrent-based data transfer. In this case, we use NWS bandwidth measurements to greedily provision Level-2 nodes to increase the fan-in, i.e., the number of nodes simultaneously transferring data to the center, to utilize the maximum center in-bound bandwidth. Table 5.3 shows the times taken to deliver a 2.0 GB file using the standard BitTorrent protocol. Compare these to the transfer times using our timely staging shown earlier in Table 5.2: both Client Offload and Center Pull in our approach out-perform by 11.5% and 6.8%, respectively, the corresponding steps in regular BitTorrent transfer. These results show that active bandwidth monitoring serves as a good technique to improve staging times.

Employing Decentralized Staging

In the above experiments, the bandwidth available between the Level-2 nodes and the center, which dictates Center Pull times, is greater than that between the client and the center, which dictates direct transfer time. Thus, the center always decided to perform decentralized staging. In the next experiment, we modified the setup to use a faster node as the client site, and repeated the experiment for staging a 2 GB file. First, we do the transfer without considering direct transfer and always using decentralized staging. Second, we repeat the experiment with the ability to choose between direct and decentralized staging depending on the ability to meet a transfer deadline (job startup). We observed that for the first case, the time to stage and transfer the data to the center was 2867 seconds. In contrast, for the second case the direct transfer completed in 968 seconds, an improvement of 66.2%. This stresses the need for the staging mechanisms to dynamically adjust to the variations in the system behavior, and to not be hard-wired to simply always do a staged transfer or a direct transfer.

Multi-Input Staging

Next, we study the ability of our decentralized staging to accommodate input data from multiple sources. We consider three configurations, shown in Figure 5.5, with two sources (X and Y) of data in addition to the client site (S). In *I*, the data from all sources is staged in a decentralized manner. This captures retrieving data from slower external sources. In *II*,

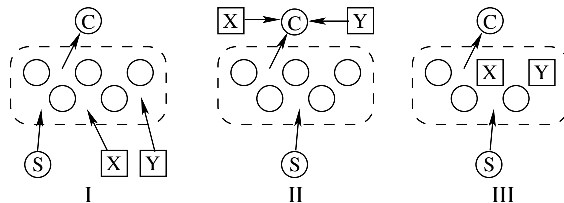


Figure 5.5: Configurations used in Multi-Input test.

we consider fast external sources, e.g., online data repositories [78] so the center can directly retrieve from them. Finally, in *III*, the intermediate nodes may already have the data, such as collaborating sites in TeraGrid jobs [19]. For each case, we compare `scp` and GridFTP from the sources to that of our staging. Table 5.4 shows the results. It is observed that decentralized staging is able to handle multiple sources, and has the potential to outperform the direct transfers by 79.3%, 90.8% and 80.9% for `scp` and 65.3%, 83.3%, 63.6% and for GridFTP, in scenario *I*, *II*, and *III*, respectively. In real transfers, the various configurations will switch depending on the transfer rates and staging deadlines.

Behavior Under Failures

Improved transfer times are key to JIT staging, and thus reducing scratch space usage times. In the following set of experiments, we study how failures will affect the transfer times under our framework.

First, we examine intermediate node failures. We focus on our decentralized staging, as a failure under direct will result in the data transfer not completing by job startup time, consequently leading to obvious job rescheduling. Figure 5.6 shows transfer time achieved by our approach under various failure scenarios, normalized to direct transfer time. We failed two intermediate nodes under three different scenarios: two Level-1 nodes fail, a Level-1 and a Level-2 node fail, and two Level-2 nodes fail. In this test, the number of replicas at each

Table 5.4: Comparison of multi-input data transfer under direct and decentralized staging.

Step	Transfer Time (s)		
	Conf I	*Conf II*	*Conf III*
`scp`	1505	1732	1789
GridFTP	901	946	934
Client Offload (S)	318	672	740
X offload	646	92	N/A
Y Offload	574	142	N/A
Center Pull	312	158	340
Staging time	312	158	340

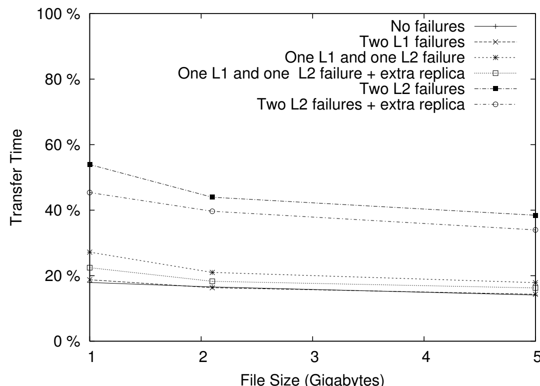
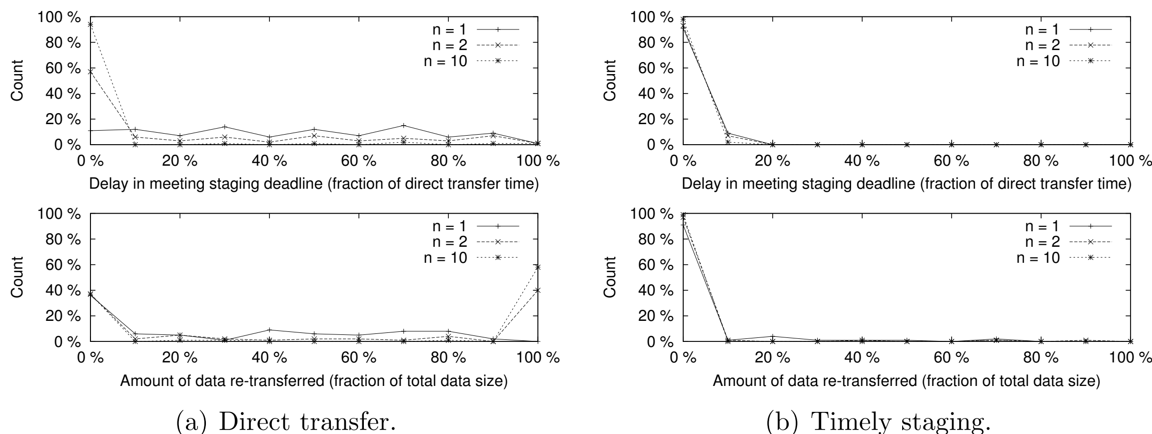


Figure 5.6: Transfer time as different combinations of Level 1 (L1) and Level 2 (L2) nodes are failed. The results are normalized with respect to a direct transfer.

level is set to 3. The system tolerates two Level-1 failures, i.e., 20% of Level-1 nodes, with negligible affect. A failure at Level-2 increases the transfer time somewhat (by a factor of 1.3), but two Level-2 failures are significantly more disruptive (time increases by a factor of 2.7). However, this is an extreme case with 25% of the Level-2 nodes failing. On the plus side, the transfer time, even with these failures, is less than half (41.2% on average) that of the direct transfer. Furthermore, our flexible design can easily accommodate extra replicas to improve fault tolerance, as observed by the reduction of transfer times for each of the Level-2 failure cases when one extra replica is used. This experiment shows that the dynamic rerouting of our approach can adapt to the changing network conditions and ensure meeting the staging deadline with minimal delays, if any. Moreover, the use of a flexible routing path between the client site and HPC center allows for offsetting delays due to intermediate node failures.

These results also imply that using the Reed-Solomon [84] error coding, described in Section 5.2, can provide an additional layer of protection when used in coordination with the replication described above. Hence, error coding at the source along with replication through multiple data flow paths can provide good fault tolerance for the staging process.

Next, we examine how failure in the scratch space affects the ability of a transfer scheme to meet a given job deadline. Here, we capture the early-transferring approach of users by starting the direct transfers as early as $T_{JobStartup} - n * T_j$, with $1 \leq n \leq 10$. Next, we randomly introduce a single failure on the scratch space between the time of starting the transfer and $T_{JobStartup}$, and determine the delay in meeting the job deadline, as well as the extra amount of data that has to be transferred. For timely staging, we assume perfect prediction, so it starts staging-in data as late as possible for a given file size. The experiment is repeated 25 times using files of sizes from 1 GB to 5 GB, for each studied n. Figure 5.7 shows the distribution of delay in meeting a deadline and the amount of data re-transferred, respectively. In the distributions, a higher count for a smaller x-axis value is desirable as that implies less delay and higher chances of meeting a deadline, and less data re-transfers.



(a) Direct transfer.

(b) Timely staging.

Figure 5.7: The distribution of staging delay and re-transmission overhead for 25 transfers with one scratch space failure. `n` represents by how early data staging is started before job startup, with higher `n` implying an earlier start of the staging process.

Our timely staging shows excellent properties with 98% of the transfers completing with no delay. In contrast, only a direct transfer that starts as early as with $n = 10$ is able to come close with 94% transfers without delay. With $n = 2$, only 31% of direct transfers complete in time. The flip side is that by staging early, the data remains exposed to the failures on the scratch and possible re-transfers. It is observed that while over 91% of the transfers in our approach had no retransmissions due to exposure to failures, that is only true for 36% of the cases with direct transfers.

Note that since we introduce a single failure, the maximum overhead is 100%. In real scenarios, multiple failures can further exacerbate the problem, as the re-transfer may now take much longer than the earlier transfer or failures in the system may prevent immediate response to a failure. This implies that delaying staging is preferable. Thus, JIT staging is able to withstand failures much closer to the job deadline, and the delay if any is small, and can be mitigated by assuming a slightly tighter deadline than actual (Section 5.1).

Table 5.5: Statistics about the job logs used by *simHPC*.

Duration	22753 Hrs
Number of jobs	80025
Job execution time	1 s to 120892 s, average 5849 s
Input data size	2.28 MB to 7481 GB, average 65.3 GB

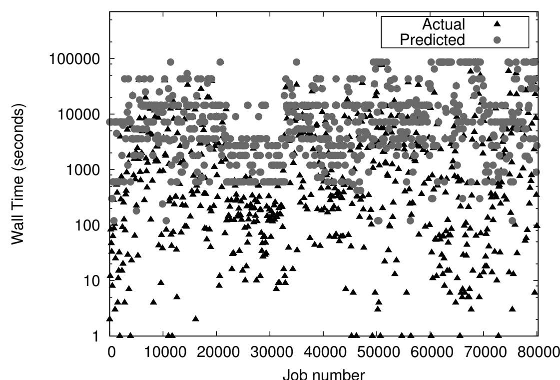


Figure 5.8: The actual and user predicted run-times for each job in the logs. Users typically request significantly more wall time than necessary.

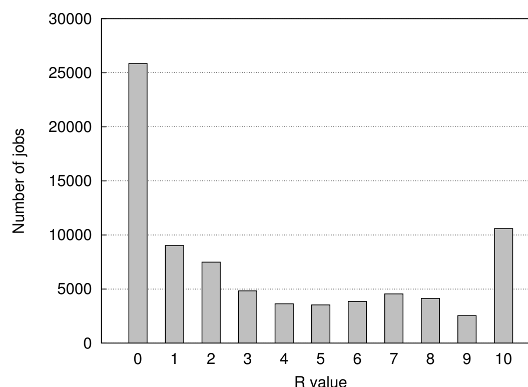


Figure 5.9: The R values for each job in the trace grouped into 10 bins. Each bin represents the fraction of the requested wall time actually used.

5.4.2 Log Analysis

In this section, we examine the three-year Jaguar [18] supercomputer job logs (Section 5.3 and Appendix A) in depth to gain information that can improve the implementation of our JIT staging service. Table 5.5 shows some relevant characteristics of the logs.

Comparing Actual and User-Estimated Job Run-times

First, we examine the accuracy of user-estimated run-times, as many works [99] have noted that users generally request more resources than required by their jobs. Figure 5.8 plots the user requested run-times with the actual run-times as recorded in the logs, and confirms this perception. Across the logs, the users over-estimated the requirements by 50.9 times on average for jobs longer than 30 seconds (430 times for all jobs), mostly due to jobs ending prematurely. Some of this discrepancy may be due to errors encountered by users while running their jobs, which is pertinent information for our staging service. Nonetheless, much of the difference appears to be users being cautious in specifying requirements, mainly to ensure that their job completes regardless of any transient issues that may occur at the HPC center. As stated earlier, this over-estimation works against our overall goals, since we would like to stage user data to the center as late as possible.

A complementary way of examining actual and user-predicted run-times is to consider the ratio of actual and user-predicted job run-times, or R value. The R values can be used to predict accurate job run-times from user-predicted run-times [99]. Figure 5.9 shows the R values for the studied trace, classified into 10 bins. For example, the 0th bin represents an R value of 0.0 to 0.09, and is equivalent to a job using 0% to 9% of its requested time. We

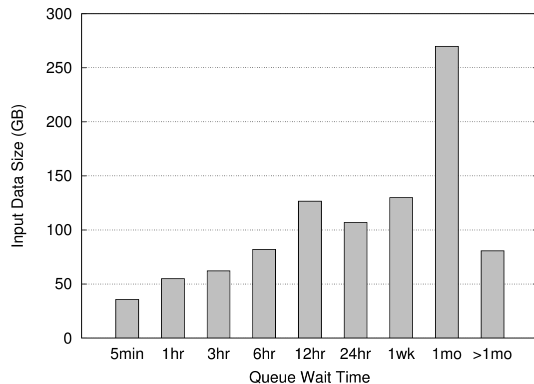


Figure 5.10: The effect of queue wait time on average input data size. Each bin value represents the maximum non-inclusive wait time for jobs in that bin.

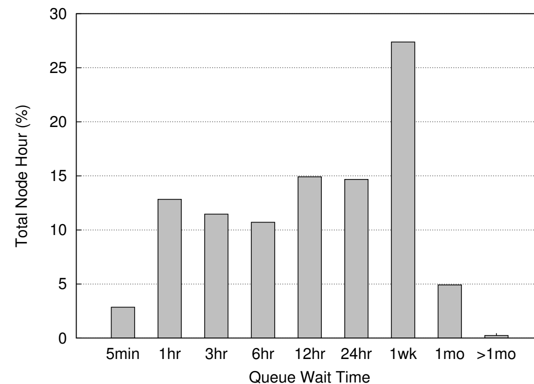


Figure 5.11: The effect of queue wait time on utilization (node hour). Each bin value represents the maximum non-inclusive wait time for jobs in that bin.

observe that most jobs do not run for their requested run-time. For example 32.3% jobs use less than 10%, and 63.5% use less than 50%, while only 16.4% use more than 90% of the requested allotment. There is also a small (but significant) number of jobs that use more than 100 percent of their predicted time. We presume that these are either completed jobs that spend a few extra seconds freeing resources, or jobs that have encountered errors. Out of 10584 (13.2%) jobs that use more than their allocation, 890 run 2 minutes past the requested allocation time and only 141 run 5 minutes past the requested allocation time.

Examining Trends in Queue Wait Times

In Section 5.1, we discussed using a batch queue prediction service to provide staging deadlines, however, the estimates provided by these services are primarily estimates of jobs' queue wait times. Queue wait times are particularly important to our JIT staging as a job must have sufficient queue wait time left for the required data to be staged in on time. Moreover, understanding the relationship between queue wait time and other important center metrics could provide further insights for refining the design parameters of JIT staging.

First, we examine how a job's input data size is associated with queue wait times. Figure 5.10 shows the result for queue wait times ranging from under 5 minutes to over 1 month. It is observed that the average input data size grows slower than the queue wait time. For example, a job that spends 3 hours waiting in the queue has an average input data size of 62.2 GB, while a job that spends 6 hours (100% increase) in the queue has an average of 31.8% more input data or 82.0 GB. This trend is similar at longer queue wait times as well.

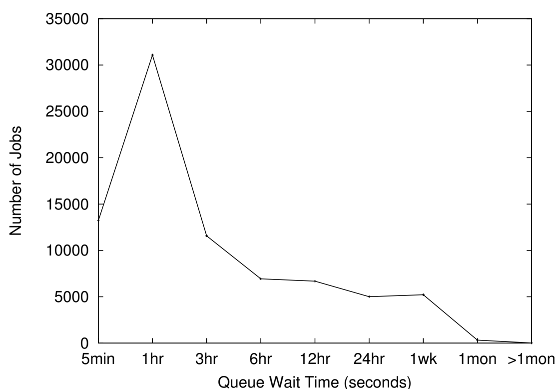


Figure 5.12: The number of jobs in each bin.

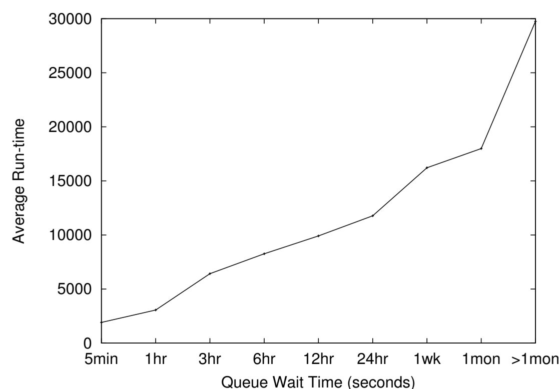


Figure 5.13: The effect of queue wait time on average job run-time.

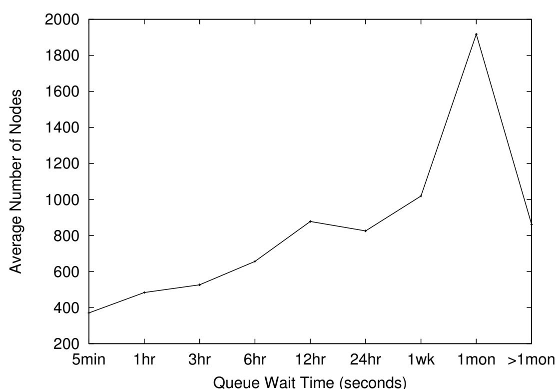


Figure 5.14: The effect of queue wait time on the average number of nodes used for a job.

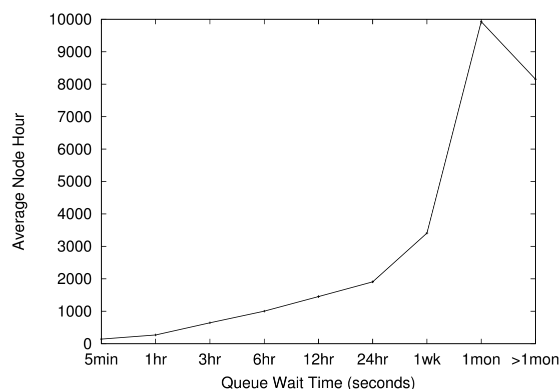


Figure 5.15: The effect of queue wait time on average utilization (node hour).

However, we note that a fundamental assumption in our analysis and simulations is that the *input data sizes = num cores * memory used*. Larger jobs frequently mean more cores and memory, but they may not always mean more data. Jobs can simply be large simulations that use very little data but produce large outputs. On the contrary, data analysis jobs are usually not that large in terms of cores and memory, but consume large amounts of input data. Unfortunately, the information in the job logs does not allow making this distinction. Nevertheless, the logs suggest that for a broad range of medium to large sized jobs, enough time is spent waiting in the queue before running to provide an opportunity to do JIT staging of job data.

Second, we examine the association of queue wait times and center utilization expressed as a fraction of total node hours as shown in Figure 5.11. Here, the *node hours* are defined as $walltime * numberofnodes$. For these logs, jobs that spend long periods of time waiting in the queue use significant center resources. The single bin with the largest amount of node

Table 5.6: The number of jobs impacted by unexpected job failures.

Time Window	Jobs Affected				
	Total	Avg.	Max.	Min.	Absolute Total
30 s	10049	1.52	20	0	9422
1 min	17686	1.89	40	0	14869
3 min	37988	3.18	210	0	24397
5 min	53964	4.09	210	0	29551

hours used is $<$ 1 week (longer than 1 day) with 17.8 m node hours or 27.4% of the total center utilization. Another interesting observation is that jobs that spend more than 12 hrs in the queue account for 62.1% of the overall utilization of the center, even though they only account for 21.5% of all jobs. This analysis suggests that jobs which use significant center resources are also the ones most able to take advantage of our JIT staging service.

Finally, a range of important trends relating to queue wait time, such as run-time, and average utilization are plotted in Figures 5.12, 5.13, 5.14, and 5.15. We note that these trends may not be applicable for other logs and HPC installations, but provide useful insights in realizing a JIT data staging service.

Quantifying the Effect of Unexpected Job Failures

Unexpected job failure may cause jobs waiting in the queue to start executing immediately, (much) earlier than their planned/predicted start times. Such unexpected job execution poses potential problems for JIT staging, as the staging of a job's associated data may have not yet completed. In this experiment, we analyze the job logs to quantify the impact of unexpected job failures by measuring both how frequently they occur and how much time the waiting jobs spend in the queue prior to running.

We step through the logs and examine each job individually. If a job does not use its entire requested allocation, we treat it as a potential failure, and observe all of the subsequent newly running jobs that start in a short time window after the initial failure. The time windows examined range from 30 seconds to five minutes. This approach assumes that every job starting in the time window was directly affected by the failing job and is likely to over-estimate the number of new jobs. We count the total number of jobs affected by failures and the average, maximum, and minimum number of new jobs per failure. The results of this analysis are shown in Table 5.6. Since each job is examined individually, there is the potential to count the same new job multiple times. To quantify the impact of such duplication, we also included the "Absolute Total," which removes any duplicate jobs that have been counted several times. On average, each job failure causes only a few new jobs to start unexpectedly, from 1.52 to 4.09 jobs. However, for the larger time windows, there can be up 210 new jobs that appear to be affected. Overall, anywhere from 11.8% to 36.9% of all jobs can be affected by job failures, depending on the time window.

Table 5.7: Input data size and amount of queue wait time for jobs affected by the failure of other jobs.

Time Window	Queue Wait (Min)			Data Size (GB)		
	Avg.	Max.	Min.	Avg.	Max.	Min.
30 s	303.4	72726	1.67	66.4	7481.7	0.24
1 min	341.2	72726	1.67	67.4	7481.7	0.24
3 min	326.4	72726	1.67	70.4	7481.7	0.24
5 min	312.6	72726	1.67	69.2	7481.7	0.24

We also examined the amount of time jobs spend in the queue and the input data sizes to see how much opportunity there is for our JIT staging service. The results can be seen in Table 5.7. The analysis converges to the top and bottom values quickly for both queue wait time and input data sizes. The average queue wait times range from 303.4 minutes to 341.2 minutes, while the average data sizes range from 66.4 GB to 70.4 GB. It seems unlikely that these are general results, but they emphasize that a JIT time staging service must be able to handle job failures.

5.4.3 Simulation Results

In this section, we use the job logs discussed above to study the performance of timely staging using *simHPC*.

Impact on Scratch Space Usage

In this experiment, we quantify the impact of timely staging on scratch space usage. We play the logs in our simulator and determine the amount of scratch used both under direct and timely staging. For this test, we assume that the scratch is empty at the beginning, and use perfect batch queue prediction. Moreover, the center is setup for weekly purges of the scratch space and the maximum center in-bound bandwidth is limited to 10 Gb/s. Only input data is considered, and a data item is only purged if its associated job has completed. Figure 5.16 shows the instantaneous savings in scratch space usage by timely staging compared to direct, measured every 10 minutes. The instantaneous savings (associated with a job input data) become zero as the job startup time approaches, as timely staging has to bring in the necessary data. A more representative aspect is the average savings over a period of time, as it captures not only the savings but the duration for which the savings were possible. Therefore, we also show the average savings calculated per hour. Finally, we calculated the average savings per hour across the entire log, and found that staging potentially uses 2.43% less scratch per unit of time (e.g. 24.9 GB/Hr on average per Terabyte of storage) compared to direct. Thus, timely staging is a promising way for conserving precious scratch resource.

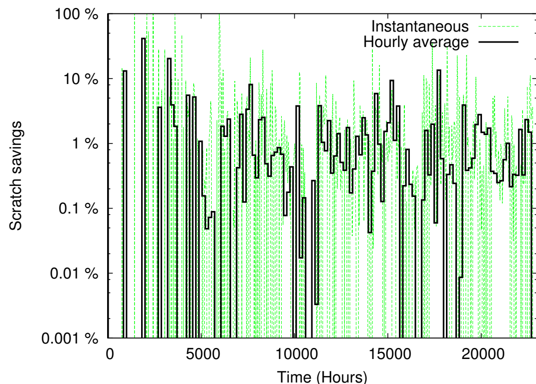


Figure 5.16: Scratch savings under timely staging compared to direct transfers. The purge period is seven days.

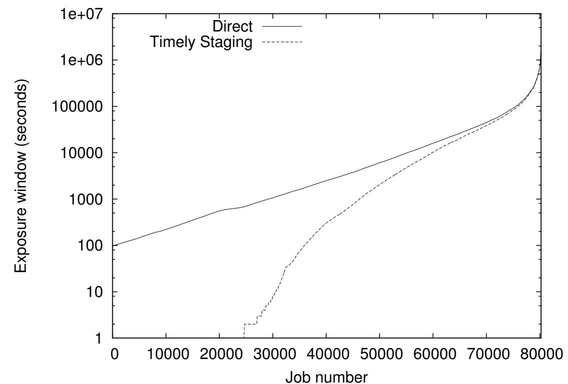


Figure 5.17: The exposure window for each job in the log under different approaches.

Effect on Exposure Window

In this experiment, we repeat the previous experiment, but now study the exposure window (E_w), i.e., duration for which the data has to wait on the scratch before the associated job is run. Figure 5.17 shows the observed E_w under direct and timely staging, for each job in our log, arranged in ascending order. In this experiment, timely staging can potentially reduce the E_w of 30.7% of the jobs to zero, and for the remaining jobs it was capable of reducing E_w by 64.2%, i.e., 75.2% reduction on average across all jobs. Moreover, we found that E_w was reduced by more than a factor of 10 for 48.3% of the jobs. However, it is seen that some jobs ($\approx$ 1.3%) with large E_ws saw only negligible ($<$ 1%) affect from timely staging. The reason for this is that: (i) many jobs require large input data, so the long duration of transfer increases the effective E_w; and (ii) many jobs in our logs arrived in bursts, and timely staging is forced to start transfers early to ensure all necessary data is available and avoid staging errors. Overall, the significantly reduced E_w for most jobs under JIT staging shows that it can provide better resiliency against storage system failures and costly re-staging.

Effect of Job Startup Time Prediction

In this experiment, we randomly introduce up to 20% variance in the batch queue prediction and the actual job start-up time. Then, we simulate the time by which timely staging will miss the actual job start-up, i.e. staging error. Figure 5.18 shows the distribution of staging error for different prediction accuracies. The results show the dependence of timely staging on the accuracy of batch queue prediction: as the error in accuracy increases from 0% to 20%, the number of jobs with no staging error reduces from 95% to 55%. However, even with increased prediction error, the number of jobs with significant delays is much less than half (30.6% of the jobs suffer a staging error of more than 1000 seconds). Note that in this

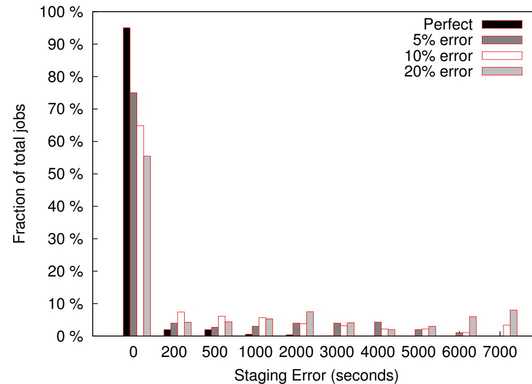


Figure 5.18: The effect of batch queue prediction accuracy on the staging error.

test, we assumed that the prediction error remains constant, however, in real scenarios, the accuracy is improved as the start-up time draws near, implying that timely staging will have much improved performance than studied in this case. Finally, the results show that the approach can withstand some prediction errors, and with improved predictions becoming available, can provide better staging alternatives.

5.5 Chapter Summary

In this chapter, we have presented the design and implementation of a timely staging framework, which attempts to coincide input data delivery with job startup. Our framework leverages periodic job wait time estimates from a batch queue prediction service, user-specified intermediate nodes, BitTorrent, and periodic network bandwidth measurements to deliver input data on time. Moreover, our prototype has been integrated with the PBS scheduler to aid in end-user adoption. Our evaluation shows a reduction in staging times compared to direct transfers, a reduction in wait time on scratch, and a reduction in scratch usage/hour. Therefore, by staging data just-in-time our framework can optimize center scratch usage, improve center serviceability, and protect input data from undesirable failure scenarios.

Chapter 6

A Cloud-Based Adaptive Data Transfer Service for HPC

High Performance Computing (HPC) is facing an exponential growth in job dataset sizes. Terabytes of reduced, result and snapshot data from experimental facilities (e.g., Spallation Neutron Source [15], Large Hadron Collider [43]), collaborations (e.g., Earth System Grid [4]), state-of-the-art cyber-infrastructure (e.g., TeraGrid [19]) and supercomputers (e.g., Jaguar [18], Kraken [11]) needs to be delivered to end-users or other destinations for local interpretation of results, visualization or for further analysis. Several applications, running on the Jaguar machine, are already producing tens of terabytes of data. Similarly, large input datasets are required to be staged into HPC centers from multiple end-user locations for consumption by supercomputing jobs. End-user data delivery services are often an afterthought in multi-million dollar HPC centers and cyber-infrastructure projects, leading to their sub-optimal use. An advanced data delivery scheme can have a significant impact on user experience and also improve HPC center serviceability.

In Chapters 4 and 5, we presented the design and implementation of frameworks that enabled the timely, decentralized, offload and staging of application data to mitigate the above issues. We focused on utilizing a group of user-specified intermediate nodes, from collaborators working on the same problem, arranged in a peer-to-peer overlay, to help HPC data transfer by providing multiple data flow paths, thereby exploiting orthogonal bandwidth between the end-users and the center. The collaborator sites provide for dynamically adjusting the data transfer by allowing data to be split and sent to multiple sites simultaneously, i.e, vary the fan-out. The sites can themselves be arranged in multiple tiers, so as to provide multiple data flow paths. Most importantly, such intermediate storage decouples the transferring of data from/to the HPC center to/from end-user sites, thus addressing the issue of end-user site availability during point-to-point transfers. However, a significant drawback of this approach is the absolute reliance on user-specified intermediate nodes, which can be quite volatile, unreliable, and scarce in all but very large collaborative projects. Therefore, a

reliable and timely data transport cannot be guaranteed through such a distributed, transient substrate. To address this, we propose to use cloud storage resources as intermediate storage for decentralized data offloading and staging.

There are many potential advantages to utilizing the cloud in HPC settings. The cloud is provisioned such that it is able to seamlessly absorb the terabytes of data emanating from simulations or observations. The cloud is also distributed with many data centers providing storage, so data can be stored in the cloud and moved closer to either the end-user or the HPC center, when appropriate. Moreover, the many data centers can be used to improve offload performance, since the data can be dynamically split and written to many cloud storage targets. Once the data is transferred to the cloud, the HPC center's storage is freed, which leaves the center less burdened and the end-user data safely stored in the managed cloud infrastructure at low cost. Meanwhile, geographically distributed researchers can access the data from the cloud storage for further analysis or visualization by staging it to their local storage. The data can stay cached in the cloud as long the users are willing to pay the costs of doing so (typically much less than the equivalent cost of storing data on center scratch space), enabling quick access for collaborators. Similarly, input data can be stored on cloud resources closer to the HPC center, thus enabling the center to pull the data from the cloud when needed.

Towards this end, in this chapter we present CATCH, a cloud-based adaptive data transfer service for HPC. CATCH provides a cloud storage framework for HPC, which utilizes proactive staging and offloading of data to cloud storage locations so as to have the input data available at the scratch storage — from multiple input sources — just before the job is about to run, and to offload output data from scratch to cloud as soon as the data is available. The goal is to reduce the amount of time that data spends on the scratch space. We utilize a combination of both a staged as well as a decentralized delivery scheme for job data. Further, we integrate CATCH with cloud resources exported by Windows Azure. CATCH seamlessly interfaces with existing cloud services, transferring data to/from the cloud, working with essentially a black box. Additionally, we export our end-user data delivery service through the file system abstraction provided by FUSE [22]. Thus, end-user programs can write and read to cloud storage and move data through them using standard file system operations. Finally, we evaluate CATCH against common HPC transfer mechanisms using our Windows Azure-based implementation.

6.1 Using the Cloud for End-User Data Delivery

Cloud computing is emerging as a viable approach for enabling fast time-to-solution for small enterprises that benefit from the cloud's pay-per-use utility computing model. The cloud supports automatic resource management, protection against data loss, and ubiquitous availability.

6.1.1 Cloud as Intermediate Storage for Data Transport

A main challenge in developing a distributed HPC center-user data delivery framework, as envisioned by CATCH, is the need for a bevy of geographically distributed storage nodes to facilitate data flow. To this end, we aim to utilize the cloud to provide intermediate storage on the path from the end-user to the HPC center, so as to facilitate efficient data transfers.

A number of cloud features make it suitable for CATCH. First, the cloud provides scalable, distributed, and always available storage. For example Windows Azure allows blobs (binary large objects), each of up to 50 GB at present [67]. Given that our approach will use cloud storage as an intermediate location during transport, even larger amounts of overall data can be handled. Thus, a wide variety of HPC applications can be supported by the resulting data delivery services. From an HPC center's standpoint, data can be stored in the cloud and only moved to expensive on-site scratch storage when needed, dramatically reducing the total amount of data HPC centers must store. From the end-users' perspective, data could be handed-off to the cloud, which frees the users from explicit data management that is typically required when using HPC resources. Second, the cloud can provide very high data reliability guarantees through replication, geographically distributed storage, and active fault ramifications. This relieves both HPC centers and end users from expensive data redundancy improving operations. Third, data can be strategically placed in the cloud, i.e., relatively close to an HPC center or end-user, yielding potentially higher transfer rates and lower latency when the data is needed. This is further enhanced if the cloud service provider supports Content Distribution Networks (CDNs). Finally, the cost of utilizing cloud storage resources is very low compared to the multi-million dollar storage systems at HPC centers. The conjoined use of HPC and cloud storage can increase the serviceability of the HPC scratch storage. This is a very attractive solution, given that HPC acquisitions are typically done on the basis of FLOPS/$ and I/O sub-systems are always resource constrained.

6.1.2 Azure Data Services

We have used the Windows Azure platform [67] for building CATCH. The following Azure features dictated our decision. (i) Azure provides a large scalable storage space for users, which matches typical HPC application needs: 100 TB per storage account, and up to five storage accounts per Azure subscription. (ii) The Azure storage service provides SLAs for up-time and correctness, and it is highly-available. (iii) Azure also provides CDN capability (currently in testing as Community Technology Preview), which can be leveraged to build efficient data placement that improves overall observable data transfer rates.(iv) The cost of Azure services is low, e.g., storage costs $0.15 per GB per Month, and thus feasible for our intended use of cloud storage in HPC data transport.

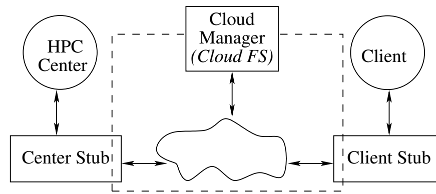


Figure 6.1: The main software components of CATCH.

6.2 Design

Cloud storage locations provide the foundation for supporting a decentralized data delivery service, e.g., for data offloading and staging, for HPC end-users. As stated earlier, the dynamic nature of the interconnects between end-user sites and the HPC center can make the amount of time it takes for a direct transfer to complete vary significantly. CATCH uses cloud storage to provide robust and efficient resources, which can be used to create on-the-fly per-collaboration/user infrastructure to support the decentralized data delivery. This helps to address the issues of purge deadlines, thus releasing center scratch storage and seamlessly moving data closer to end-users.

6.2.1 Design Overview

CATCH has three main software components as shown in Figure 6.1: *client stub* to allow for interfacing with cloud resources; *cloud manager* (e.g., a cloud file system) to interact and affect how data is stored and moved in the cloud; and *center stub* to provide a transparent interface to accessing and storing data on cloud resources.

A user who wants to run an application at the HPC center, first queries the *center stub* to get an estimate of when the user's job will be scheduled. Based on this estimate and the size of the input data, the *client stub* then determines whether a direct transfer would be sufficient. If not, the user attempts to utilize the cloud resources to facilitate a decentralized data transfer. Since storage in the cloud is cheaper compared to storage at the HPC center, such a decentralized transfer can be initiated much earlier than a direct transfer. HPC center storage is precious and user data is constantly purged to make room for data from new incoming jobs. Thus, the end-user site utilizes our software hooks to transparently move the data into the cloud. Next, we either count on the cloud internals or our pre-staging interface, through the *cloud manager*, to move the data to cloud sites closer to the HPC center. When the job is about to be scheduled, the *center stub* pulls the data from the cloud to the center PFS, thus completing the transfer. Conversely, when the job finishes (or has intermediate data for the user), the *center stub* pushes the data onto the cloud resources. The client can then retrieve this data when and how it wishes. This design essentially decouples the center-side and client-side transfers and provides flexibility and fault-tolerance. Figure 6.2 shows the high-level flow of data in CATCH.

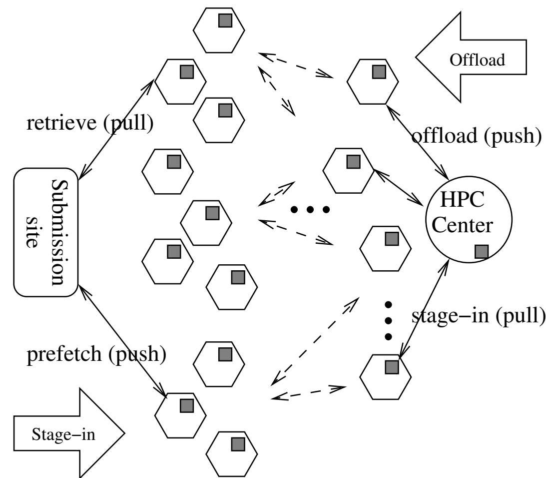


Figure 6.2: The data flow path from the HPC center to the end-user site. The intermediate resources are represented by hexagons. The gray squares represent software hooks/APIs that CATCH uses to control the data flow.

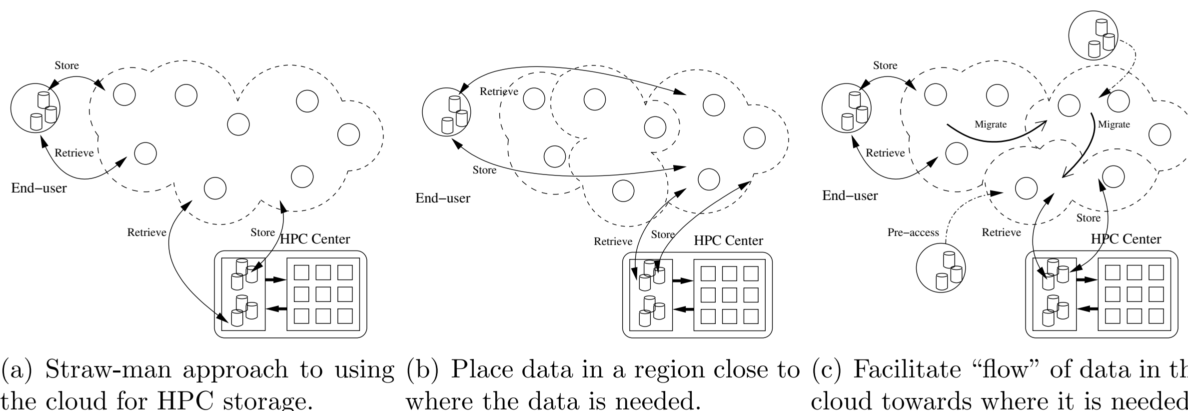
We have developed detailed models for building end-user data delivery services with ad-hoc resources in our previous work [72, 74] and in Chapters 4 and 5. In CATCH, we overcome all of the issues arising from such ad-hoc intermediate sites, by leveraging cloud resources and integrating the cloud model with HPC data movement. However, the fundamental issues of node selection and meeting delivery deadlines are very similar, thus we leverage our previous work in CATCH to this end.

6.2.2 Cloud Data Interface

The cloud provides a suitable platform for developing and expanding end-user data delivery services. We have built our software using the Azure [67] platform. In the following, we discuss several possible heuristics for the HPC data transfer system to interact efficiently with the cloud.

Straw-man Approach

The first approach that we consider is a simple use of cloud resources for storing HPC data. This straw-man approach is illustrated in Figure 6.3(a). Here, end-users push their job's data to the cloud, which can then be retrieved by an HPC center before the end-users job will run. Upon job completion, the HPC center can take the result data and store it in the cloud for the end-user to retrieve as necessary. This method uses the standard Azure API and relies entirely on the cloud for performance. For example, if the cloud either stores or moves the data closer to the HPC center, better performance would be observed. However,



(a) Straw-man approach to using the cloud for HPC storage.

(b) Place data in a region close to where the data is needed.

(c) Facilitate "flow" of data in the cloud towards where it is needed.

Figure 6.3: Different approaches for using the cloud to implement end-user data delivery services.

if data is stored at arbitrary locations, no performance improvement guarantees can be made. Nonetheless, this approach is the key step in decoupling the end-users from the HPC center, thus allowing the end-users to be intermittent and freeing them from issues of data retransmission, and resulting job rescheduling.

Utilizing Storage Regions

The main drawback of the Straw-man is that it does not exploit the data flow information, i.e., where and when a data item is needed, which is available in HPC job scripts. Moreover, typical HPC data, especially input data is stored once by the end-user and retrieved once by the HPC center, thus giving cloud management little opportunity to identify data access hotspots and migrate data to resources closer to where the data is being accessed. Thus, Straw-man cannot ensure that cloud-enabled decentralized data transfer would yield better transfer performance compared to a point-to-point transfer. However, transfer rate performance gains are desirable when retrieving data at the HPC center as delays may cause the associated job to be rescheduled, consequently increasing job turn-around time and affecting overall center serviceability.

To address these issues, we exploit Azure's support for specifying regions for storing data to reduce data access latency experienced by the HPC center. This approach is illustrated in Figure 6.3(b). Here, the end-user can choose to put the data in a particular part of the cloud that is closer to the HPC center. In this use-case, although the end-user may want to (eventually) store data on resources that are farther from her site (closer to center), the Azure management may hide the increased transfer latency from the end-user by allowing data to be placed nearby and then migrating it to the specified region transparently. If such support is not available and higher transfer latencies are exposed to the end-user, the user can choose to transfer the data into the cloud much earlier to avoid delays and potential job

rescheduling if the HPC center needs the data before the transfer is completed. Based on our interactions with HPC users, we note that most users tend to start their data transfers well in advance (sometimes on the order of days). However, the cloud provides a better option for advance transfers compared to moving data (well before job startup) to the precious PFS on the center, where it can hinder the center's ability to service other currently running jobs.

One challenge in implementing this approach is how the region specification can be made transparent to the application. To this end, we assume that: (i) the end-user (via collaborators) has multiple storage accounts in different regions in the cloud; (ii) information is available in the job script (as discussed earlier) to determine which region a data item should be stored in; and (iii) the information can be relayed to CATCH runtime. The runtime can then utilize appropriate Azure API to accomplish region specific storage. We note that region specification in Azure seems to be static, thus a priori knowledge of where the data would be consumed is needed. That said, most data consumption locations can be derived from the HPC job scripts, so this is not expected to be problematic. A bigger challenge is that the granularity of the regions available in Azure is too coarse, e.g., only a handful of regions (South Central US, North Central US, and Anywhere US) are available for the entire US. This could limit our ability to derive optimal performance from our data transfer service.

Facilitating Dynamic Data Flow

Our main goal is to develop a service that allows data to "flow" closer to locations where it is needed, before it is accessed, so as to reduce access latency. CATCH can benefit if individual cloud storage locations and their performance were known. However, the key cloud advantages of transparency and decoupling of management from usage, pose a hurdle for our approach. A set of distributed cloud storage resources are not as configurable as a set of explicit collaborator sites providing storage (such as those explored in our previous work on decentralized HPC data transfers).

The CDN service provided by Azure can yield a more dynamic and robust data transport than using regions, e.g., by caching data close to the HPC center. This is useful for output data from HPC jobs as it would be consumed by many collaborators, which provides enough repeat accesses to the same data to enable the cloud mechanisms to optimize data placement. However a CDN only improves performance for repeat accesses, and as stated earlier, HPC input data is often consumed once. We overcome this problem by utilizing collaborator sites closer to the HPC center to pre-access data before it is retrieved by the center, potentially triggering CDN-enabled data migration. This will move the data closer to the collaborator site. Since, the HPC center (or conversely end-user) is also nearby, the intuition here is that accesses to the data from the center when needed will thus experience lower latency. This approach is illustrated in Figure 6.3(c). In essence, by accessing the data from collaborator sites closer to the center, the data is prefetched to high speed CDN locations, making it

readily available for the center when needed. Note that pre-accessing the data does not imply downloading the entire dataset. Rather, reading a few random bytes in a blob is expected to do the trick (as the blob is treated as a monolithic unit for CDN purposes), without incurring the cost of reading large data from the cloud.

An alternative approach to using the CDN capability is to leverage the availability of multiple cloud accounts, e.g., belonging to different collaborators and in cloud regions that are close to them. The relative distance of collaborators (in terms of available bandwidth) can be determined using standard network monitoring, e.g., NWS [107], and the collaborators are then arranged on the end-to-end path from the user to the HPC center. The end-user can then store the data into his account and pass the appropriate access credentials to the collaborators, who can then invoke copying of data from one account to another. This would in essence move the data closer to the HPC center. We note that this is a non-standard use of the cloud API. However, the advantage is that this approach allows for explicit monitoring, and can affect the flow of data through the cloud at much finer granularity, consequently, leading to improved HPC data transport.

6.2.3 Data Transport as a File System

In order for the entire end-user data delivery mechanism, through the intermediate cloud storage nodes, to be transparent both to the user as well as the HPC center, we put forth an easy-to-use file system interface. In our design, the client and center stubs talk to a transparent file system mount point, provided through FUSE [22] as *Cloud FS* (Figure 6.1), which abstracts the process of accessing the cloud storage and in addition moves the data closer to the end-user or the HPC center. The use of FUSE to abstract access to different storage substrates has gained wide spread popularity due to the ease with which purpose-built storage systems can be transparently made available by having them implement certain POSIX APIs (e.g., s3fs [25] for Amazon S3 or stdchk [56, 62], a file system atop distributed storage made of disks, memory, or SSDs.) The `read()` or `write()` call in these situations typically abstracts parallel striping or a network transfer, respectively. The novelty of our approach in *Cloud FS*, however, lies in the fact that we hide the data transport behind a file system interface.

We have developed a scalable and robust FUSE-based *Cloud FS* module to allow end-users and HPC center management tools to access cloud storage. An in depth discussion of *Cloud FS*, and how we use it to access cloud storage is presented in Section 6.3. Here, we focus on how the file system abstraction can serve to capture cloud data flow. To this end, we augment the FUSE driver semantics with the notion of *data flow*, in addition to the basic get and put services (i.e., a `write()` call will also need to implement methods necessary to propagate the data further in addition to the standard network transfer required to store the data in the cloud.) Such an approach not only allows us to store data into the cloud, but also helps to migrate the data towards its final destination.

```
#PBS -N myjob
#PBS -l nodes=128, walltime=12:00

mpirun -np 128 ~/MyComputation

#CollabAcct collab1.blob.core.windows.net:50GB
...
#CollabAcct collabN.blob.core.windows.net:30GB
```

Figure 6.4: An example annotated job script with cloud storage specific directives.

The augmented module performs a number of functions. (i) It has to negotiate access to the cloud storage. This is achieved by providing *Cloud FS* a list of account credentials at start up. This can be a single account or a list of credentials to be used appropriately. To allow users to control what credentials to use for data accesses through *Cloud FS*, we provide an `ioctl` call to specify the identifier of credentials to use. The credentials to use can be changed as often as before each data access. However, typically the module will automatically determine which account/region/location to use as per the data transfer SLAs. (ii) The module stores and retrieves the associated data chunks from the cloud. To facilitate this, the source of the data, i.e., the HPC center stub in offloading and client stub in staging, maintains a mapping of dataset to chunks (and their locations in the cloud). On an offload from the center, the client stub can use the mapping information available at the center stub to pull the necessary chunks of the datasets from the Cloud FS. Similarly, for a staging from the end-user, the client stub provides the location of the input dataset chunks. (iii) The module may also have to probe different intermediate locations to determine the best path to utilize. One approach is to perform a number of small GET and PUT operations on the cloud, and determine observed bandwidth, which can then be used to select appropriate storage regions. Another approach, if the cloud service provider supports it, is to use cloud monitoring services. This information can then be used transparently to change storage regions and achieve better flow rates. The module integrates such interactions into the data flow, thus providing transparent services to the users. Using the aforementioned FUSE-based data flow file system, the client and center stubs can orchestrate the cloud intermediate nodes into multiple levels to move the data closer to the destination.

6.2.4 HPC Job Submission Integration

We propose to specify the cloud accounts and those of the collaborators as part of the user's job submission script (e.g., PBS [32]). Special directives can be used to annotate the job script as shown in Figure 6.4. This way, the cloud storage sites associated with the collaborators become an integral part of the job and can be used by the center stub for the end-user data delivery. End-users can further qualify the job submission scripts with usage properties of the collaborator's account, e.g.,how much storage to make available or what is the load threshold. This information can then be used by the stubs to derive how best to route data between each other.

6.2.5 Viability of Using Cloud Resources

An important consideration in the design of CATCH is the cost of utilizing cloud resources. For example, transferring tens of terabytes of data through the cloud multiple times during the life of a single job may result in excessive cloud charges, as cloud service providers often bill per unit data transferred and stored. However, a number of factors work in the favor of CATCH-like systems. First, the cost of using cloud resources is falling sharply [68], and wider adoption of cloud resources is likely to continue this trend. One can argue that the amount of data being used is also growing, and thus the impact of falling prices may be negated. We note that the increase in data impacts both centralized provisioning on HPC centers and cloud resource provisioning similarly, and although crucial, should not be a deciding factor in this context. Second, much like how cloud computing is seen as a viable alternative for mid-sized computing (e.g., jobs requiring a few thousand cores), there is also a tipping point up to which cloud storage is viable for HPC job data. We analyze these scenarios in our evaluation. Further, our analysis of three years worth of logs from the Jaguar supercomputer [74], shows that there exists a large number of jobs that are mid-sized, and do not involve terabytes of data. These jobs can benefit from CATCH. Finally, the upfront costs of I/O management and acquiring disks for large supercomputers may easily exceed tens of millions of dollars. While, such costs can certainly be amortized over the lifetime of an HPC center, it still cannot retain job data beyond a certain window of time. CATCH provides a way to complement such storage at relatively low costs without high upfront costs.

6.3 Implementation

We have implemented CATCH using about 2500 lines of C# code with the Windows Azure [67] platform as the cloud storage backend. Although our current implementation utilizes Azure, our design is general enough to be interfaced with other cloud service providers.

6.3.1 Architecture

The main components of CATCH are shown in Figure 6.5. The *Cloud FS*, supported via FUSE, provides applications with a transparent interface to CATCH. Once the application data is written to a PFS, the center stub simply writes that data to a special mount point, or performs `ioctl` calls for control commands, and the *Cloud FS* component converts data access to operations on the cloud. The *Scheduling Monitor* interacts with the center wide job scheduler or with Batch Queue Prediction (BQP) [7] to determine when a job completes or when it is likely to run. This information is reported to the *Transfer Manager*, which uses it to determine when to start a transfer. The *Network Monitor* determines what cloud accounts provide the best transfer rates by occasionally PUTing or GETing test blobs to

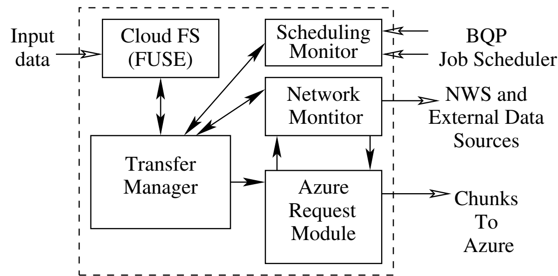


Figure 6.5: Architecture of CATCH.

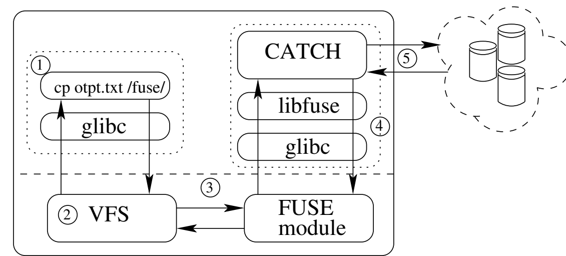


Figure 6.6: Architecture of *Cloud FS*.

the cloud and measuring bandwidth. The *Transfer Service* component uses the bandwidth and scheduling information to decide where and when data should be stored to or retrieved from the cloud. The *Transfer Service* then splits the data into chunks and passes them to the *Azure Request* module. This module is responsible for interfacing with the cloud storage service and creates the appropriate HTML requests.

FUSE Module Interface: The architecture and the flow control in our FUSE-based *Cloud FS* module is displayed in Figure 6.6. When an I/O operation is performed on a file in our mount point (step 1), it is redirected to the *Cloud FS* module (2, 3, 4) by the FUSE runtime. *Cloud FS* then processes the I/O to take appropriate cloud actions (5).

6.3.2 Data Operations

Access to the Windows Azure [67] Blob service is achieved through a RESTful API, where all operations are performed using HTTP requests. The Blob service provides two types of blobs. (i) Block blobs are made up of “blocks” that can each be of up to 4 MB, a block blob can have up to 50,000 “blocks” providing a maximum blob size of 200 GB. Block blobs have commit-update semantics, i.e., a number of blocks are first uploaded, and then another request is sent to commit the changes. (ii) Page blobs, that we use in CATCH due to their similar semantics to standard files, consist of 512-byte regions and provide immediate or in-place updates much like a traditional disk. They also have a large maximum blob size of 1 TB. Each CATCH operation corresponds to HTTP requests with particular query, headers, and other parameters. In the following, we describe different data functions supported in CATCH.

Storing Data

This operation is achieved using an HTTP PUT request. In addition to the content of the data, the request also includes information about the content length and the particular region of the blob to write to, i.e., offset. With page blobs, the maximum request size is 4 MB (consisting of contiguous 512 byte aligned regions). Files larger than this are broken into chunks before being sent to the cloud storage. Upon receiving the HTTP request, Azure parses the request and stores the data.

Retrieving Data

Retrieving blob data is the inverse operation of storing data, and is done using an HTTP GET request. This request must also include a region of the blob to read. If the request succeeds, the body of the response will have the requested data. CATCH tries twice to read the data before reporting an error to the application. The retry mechanism is added to overcome trivial failures due to lost messages or delayed response from Azure.

Data Flow

Center and client stubs need to use the Cloud FS to orchestrate data flow. To provide these stubs with options to control different cloud functions, we have enabled a set of control knobs that can be set using `ioctl` calls. The FUSE layer exposes a set of POSIX APIs that any underlying system can implement to provide access to its features. Unlike `read()` and `write()` calls, `ioctl()` allows us to manipulate the underlying device parameters of the special files. This provides us an elegant way to orchestrate many sophisticated data flow operations on the cloud storage, much beyond basic store and retrieve functionality.

Consider a case where the HPC center stub wants to disseminate the chunks of a dataset to different geographic regions to facilitate better data access to end-users. The center stub first interacts with the client stub to determine an appropriate data flow path. Let us assume that three regions, R_1, R_2, and R_3 can provide the best data flow, and the credentials to use the regions are already available to the *Cloud FS*. Before data is written to the cloud, the center stub issues a "SET REGION" `ioctl` to *Cloud FS* to indicate that the data should be written to R_1, which is done when the data is written. Then, CATCH decides that data should be moved to R_2. This copying of data is initiated through a "COPY REGION" `ioctl` to *Cloud FS*. This indicates that cloud service calls for moving data from R_1 to R_2 should be issued. Another "COPY REGION" call can also be issued to move the data to R_3. This completes the data flow. The stubs can also use the interface to pass a pointer to a configuration file or a structure containing one or more account credentials. This information is passed to CATCH and used for future data operations on a particular dataset or mount point. Additionally, in the default configuration, CATCH will measure bandwidth using probes to determine the fastest accounts, but instead the client stubs may specify in a structure the fraction of data to go to each account.

6.3.3 Real World Considerations

There are several factors that affect CATCH when it is used to offload and stage data. Behind the *Cloud FS* mount point, CATCH is utilized in coordination with the center PFS as part of an integrated data service. This allows for the HPC jobs to continue without being affected by the response times of CATCH's cloud interactions. In this scheme, data is

transferred from the cloud resources to the PFS before its associated HPC applications are scheduled for execution. Similarly, the jobs output data is buffered on PFS, which is then offloaded to cloud sources asynchronously from job execution and on-line data accesses.

Multi-Input Staging and Multi-Output Offloading

Our implementation is capable of retrieving data from more than just cloud resources, e.g., national data repositories, etc., and these other resources can also be incorporated into the decentralized transfer. The data sources are provided as links in the job-submission script. The transfer manager, through the network monitor, uses small scale tests, e.g., partial download or upload from a web repository, to determine expected transfer times and make staging and offloading decisions. In case of staging, the goal is to ensure staging of all input data from all sources completes before the predicted job startup time. For offloading, the goal is send the data as quickly as possible, so in the event that the additional resource is slow, it will not be utilized.

6.4 Evaluation

In this section, we present an evaluation of CATCH using the implementation described in Section 6.3. We also compare our results to popular direct transfer techniques that are the default approach for transferring data in many HPC centers.

6.4.1 Implementation Results

For our implementation experiments, we use the Azure Cloud storage service to study the effectiveness of our end-user data delivery service in a true distributed environment. We created 5 Azure storage accounts in the following regions: Anywhere US, North Central US, South Central US, Anywhere Europe, and Anywhere Asia. While there are a few more regions provided by Azure, this selection provides a representative and geographically dispersed testbed for our experiments. For the following experiments, we only consider one explicit level of intermediate storage accounts, i.e., data is pushed from the source (either HPC center or end-user site) into the cloud, and is then pulled from the cloud onto the destination. In contrast, multiple levels are created when data is moved between different accounts before being transferred to the destination. The setup consists of the HPC center, the cloud accounts, and a client. The roles of the HPC center and the client are provided by a lab machine at Virginia Tech and a remote node running on Amazon's EC2 [20]. For all experiments data is pushed to the cloud, by either the HPC center or client and then retrieved by the other role. In the following, the presented results represent averages over a set of three runs.

Table 6.1: Average observed bandwidth and transfer times for a 4 MB probe to different Azure regions.

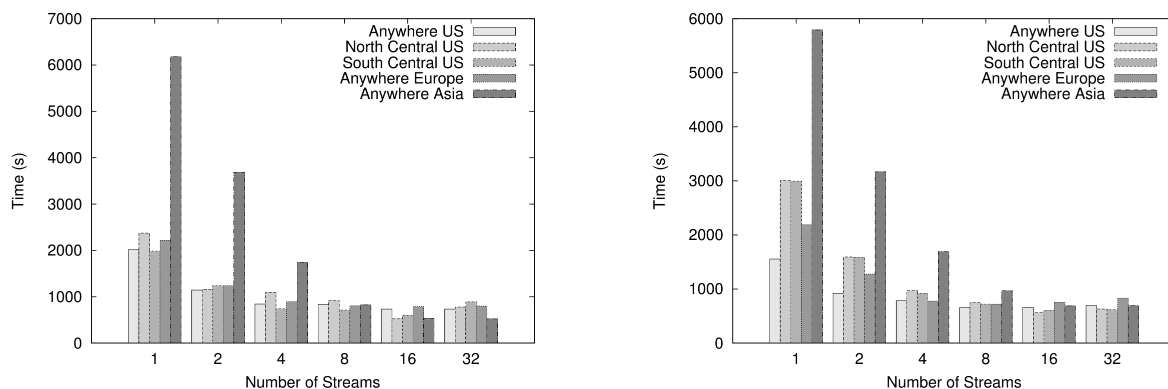
Regions	Anywhere US		North Central US		South Central US		Anywhere Europe		Anywhere Asia	
	(Mb/s)	(s)	(Mb/s)	(s)	(Mb/s)	(s)	(Mb/s)	(s)	(Mb/s)	(s)
Put	4.7	7.2	4.2	8.0	4.9	6.9	3.4	10.0	2.5	13.2
Get	5.4	6.2	3.2	10.5	6.2	5.4	4.2	7.9	1.7	19.8

Probes to Cloud Resources

A crucial component of CATCH is the ability to dynamically adjust to changing network capabilities as data is staged or offloaded. Since the cloud is a black box, we determine the best sites for storing the data by directly measuring the data rates we can obtain. In our first experiment, we determine the effectiveness of probing the cloud. To this end, we send a 4 MB dummy blob to each of the regions considered in this study, and measure the time it takes to either PUT or GET the blob. The results are shown in Table 6.1. We make two observations from the results. (i) There is a marked difference between the measured test blob access times to different regions. This is promising as CATCH can use such measurements to guide its data transport, without worrying about Azure hiding such details. (ii) This also shows that the regions are in fact distinct and provide different throughputs. Thus, if data is moved to a region closer to its final destination, better transfer times will be observed on the on-demand data access. Overall the transfer times and transfer rates to data centers in the US provide the best probe times for our location. The data center in Europe is sometimes faster, but in the worst case it is only slightly slower. From our location the slowest region is in Asia. This is expected as this region is provided to be primarily accessed by people close to Asia.

Effect of Multiple Transfer Streams on Access Times

During our previous experiments we observed that Azure is capable of handling many simultaneous requests, which can provide high aggregate throughput. To take advantage of this ability we designed CATCH to utilize multiple streams for transferring data simultaneously. In our next set of experiments, we demonstrate the effect of using multiple streams on transfer rates from Azure. In this experiment, only one region is used for each transfer, and the number of simultaneous streams varies from 1 to 32. The file size used for testing the transfer rate is 1 GB. Figure 6.7 shows the times for data transfer from client to the cloud (Write (a)), and the cloud to the HPC center (Read (b)) under different numbers of simultaneous streams. Compared to a transfer with a single stream, the multi-stream staging and offloading can reduce the last-hop transfer times by up to 88.1% and 91.3% for reading and writing, respectively. Another observation is that using between 8 and 16 streams offers the best performance overall for our setup. Utilizing more streams results in a bottleneck on our emulated end-user site and HPC center. While this result will hold for a typical end-user site, we believe the HPC center can use many more simultaneous streams without suffering from performance degradation. This is promising in that it shows that CATCH can help reduce the data staging times even more when used at a real HPC center.



(a) Data staging times from end-user site (Write). (b) Data retrieving times from HPC center (Read).

Figure 6.7: Transfer times (in seconds) to different cloud regions, using increasing numbers of streams. The file size used is 1 GB.

This result also implies that using multiple streams can delay copying of data to and from scratch space by a factor of 4.3 on average across the studied stream counts, and still get the data to the center in time for the job to start. Thus, it reduces the time the scratch space has to hold the data before it is used, consequently, improving center serviceability.

Effect of Using Multi-Region Access

In our next experiment, we repeat the probe-test from our first experiment but for accesses to multiple regions. Table 6.2 shows the times taken to transfer a 1 GB file to 2 and 3 different regions using CATCH. Data movement between the different regions is accomplished in CATCH by orchestrating data flows between different accounts through the FUSE abstraction. CATCH actively probes the cloud regions before and during the transfer to determine the fastest regions. In these cases, regions in the United States were utilized due to their higher bandwidth. The experiment was performed using 8 and 16 streams since both of these scenarios provided good performance in the previous experiment. Compare these to the transfer times shown earlier in Figure 6.7: multi-region Write and multi-region Read out-perform the standard write and read significantly (up to 43.8%) for all regions except for North Central US which has very similar times. This performance improvement was very consistent across runs. These results show that active bandwidth monitoring provides a good tool for improving transfer times.

Table 6.2: The time to transfer a 1 GB file using multiple regions.

Number of regions	Threads			
	8		16	
	Write	Read	Write	Read
2	547	544	520	548
3	592	593	588	672

Table 6.3: Comparison of decentralized transfer times (in seconds) with different direct transfer techniques. The buffer size for IBP, GridFTP, and BBCP is set to 1 MB. The number of streams in GridFTP, BBCP, and CATCH is set to 8, 16, and 16, respectively.

CATCH	
Write (Offload)	520
Read (Pull)	548
Direct	
scp	2821
IBP	1791
GridFTP	722
BBCP	573

Comparison with Direct Transfer Methods

For our next experiment, we utilized our 5 cloud storage accounts coupled with 5 PlanetLab [3] nodes to create a distributed testbed for comparing different HPC data movement techniques with CATCH. A more detailed description of our PlanetLab experimental setup can be found in Chapters 4 and 5.

We compared several point-to-point direct transfer tools that are prevalent in HPC: (i) scp, a baseline secure transfer protocol; (ii) IBP [83], an advanced transfer protocol that makes storage part of the network, and allows programs to allocate and store data in the network near where they are needed; (iii) GridFTP [34], an extension to the FTP protocol, which provides authentication, parallel transfers, and allows TCP buffer size tuning for high performance; and BBCP [21], which also provides high performance through parallel transfers and TCP buffer tuning. Note that these protocols are all typically supported [24] by HPC centers such as Jaguar [18].

Table 6.3 shows the result. One important point to note here is that while direct transfer methods include the flow of data from source to destination, CATCH Read and Write numbers only include either storing the data into the cloud or retrieving the data from the cloud. In the best case for CATCH, a Read can begin as soon as initial part of a dataset becomes available by a Write. The overall end-to-end transfer time can then be calculated as maximum of Read and Write times: 548 seconds in our case. In the worst case, there may be an arbitrary wait between the Read and Write operations. However, from the center point of view, only the time it has to stay engaged in the transfer is critical, as communication between the cloud and the end-user site is decoupled from the center. Current point-to-point transfer tools cannot enable this behavior as they expect a significant resource commitment from end-users and HPC centers for the duration of the transfer. Thus, only the access times to/from cloud are of concern. It can be observed that from this perspective, CATCH is able to achieve 6.8% (wrt. BBCP) to 81.1% (wrt. `scp`) better performance compared to direct transfer mechanisms on average across both Read/Write operations. These results suggest that CATCH is a viable option for HPC end-user data delivery services.

Table 6.4: Current Azure pricing.

Storage	$0.15/GB
CDN	$0.15/GB
Transfer	$0.10/GB

6.4.2 Cost of Cloud Usage

In the next experiment, we determine how the cost of cloud services impact CATCH usage. Table 6.4 shows the current pricing structure used by Azure [68]. Table 6.5 shows three different usage scenarios for HPC application workflows, and the cost for using CATCH for the applications. To give a sense of the scale of the job that produces terabytes of data, consider that a 100,000-core run of GTS fusion application on Jaguar produces a 50 TB dataset. Since the pricing for cloud usage are expected to fall, the Table also shows the cost of using CATCH if the prices are reduced by 10%, 50%, and 90%. In contrast, consider that in a typical HPC center, I/O subsystem costs can account for 20% to 30% of the acquisition cost and may run into millions of dollars. Even though the acquisition cost is amortized over the life of a machine, the annual running costs can still run into millions of dollars. While such PFS storage is needed at the center for a quick dump of job data, it cannot retain the data beyond a purge window, let alone the duration of a collaboration. Thus, CATCH provides a way to complement storage at the HPC center, especially for mid-size HPC applications.

6.5 Chapter Summary

In this chapter, we have presented the design and implementation of a decentralized end-user data transport service, CATCH, for HPC. The novelty of our approach lies in the transparent use of cloud storage resources as intermediate nodes, and bringing such resources to bear on

Table 6.5: Cost of using CATCH for different workflows under varying pricing structure.

	A	B	C	D
Data size	50 TB	10 TB	1 TB	500 GB
CDN usage	Yes	No	Yes	No
Num. uploads	1	1	1	1
Downloads	10	10	5	10
Cost (current)	$70,000	$12,500	$900	$625
Cost (90%)	$63,000	$11,250	$810	$563
Cost (50%)	$35,000	$6,250	$450	$313
Cost (10%)	$7,000	$1,250	$90	$63

the timely problem of HPC data delivery. To this end, CATCH provides a FUSE-based file system abstraction to the cloud storage, so that end-users can access CATCH transparently. Using this backdrop, we present several techniques to improve the end-user data delivery experience, by bringing data closer to the HPC center or the user so the data can be pulled eventually as needed to coincide job startup or a workflow deadline, respectively. CATCH exploits several desirable characteristics such as disseminating chunks to a geographically distributed set of locations, and extends them further, all in a seamless fashion to HPC users. Our results indicate that CATCH is able to exploit orthogonal network bandwidth and adapt to network conditions, reduce scratch space consumption, and mitigate the high cost of HPC I/O acquisition, especially for mid-size HPC workflows.

Chapter 7

Conclusion

7.1 Dissertation Summary

This dissertation presents the design of an Integrated End-User Data Service for High Performance Computing (HPC) centers. Our service attempts to address the issues of exponentially increasing data sizes and ad-hoc data management, through comprehensive scratch management which can provide benefits for both the center and the end-user. Our strategies comprise a novel workflow-aware caching system wherein the scratch space is treated like a cache and end-user workflow information is used to make informed data movement decisions. Moreover, we provide sophisticated end-user data delivery services, which provide an intermediate staging ground for end-user data, allowing it be staged to or offloaded from the center scratch as part of the job workflow. These combined strategies allow data to be managed without significant end-user interaction, prevent precious scratch space resources from being wasted by storing stale data, and allow end-users to potentially avoid the effects of purge policies and scratch space failures. Below, we detail the specific contributions made by each component of our Integrated End-User Data Service.

We present [73] a novel model for managing an HPC center's scratch space, "Scratch as a Cache", where the scratch is treated like a cache rather than a normal storage system. This approach limits unnecessary direct end-user interaction, and allows caching policies to manage the scratch based job workflow needs, rather than end-users' desires. 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 a range of Tier 2 storage types 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.

A crucial component of the Scratch as a Cache model is the ability to populate and evict from the cache, or scratch space. End-users' data must be frequently moved in to the cache from remote sources such as repositories or experimental facilities, processed at the center, and the resulting data must then be evicted to remote sites for analysis. In reality, population and eviction are large data transfers and correspond to data staging and data offloading, respectively. In this dissertation, we have presented both data staging and offloading services, here we discuss the specific accomplishments of our end-user data delivery services.

We design [70, 72, 76] a combination of both a staged as well as a decentralized offloading scheme for job output data, which makes use of distributed intermediate sites. Our approach presents a fresh look at offloading by using a set of user-specified intermediate nodes to construct a p2p network and transferring data based on bandwidth-adaptation. Compared to a direct transfer, our techniques have the added benefits of resilience in the face of end-resource failure and the exploitation of orthogonal bandwidth that might be available in the end-to-end data path. Our results indicate that our offloading approach improves the rate at which the data is offloaded from the center (90.4% for a 5 GB data transfer), while allowing the submission site to pull the data as and when the site becomes available, at a much higher transfer rate because the result-data has already been staged closer. Further, offloading enables us to deliver data based on a previously agreed upon SLA, dynamically varying the fan-out as necessary. An analysis of our approach using a realistic simulator, driven by a three-year log from an actual supercomputer reveals that it is better able to manage the scratch space and reduce job delays. Thus, our scheme can be extremely useful to both HPC centers and users.

We also present [71, 74, 77] a JIT staging framework that attempts to have the data available at scratch, from multiple input sources, just before the job is about to run. The framework proactively brings the data to intermediate storage sites on the path from the end-user site to the HPC center. This reduces the time for copying the data to scratch, thus providing better opportunities for JIT staging. Our framework leverages periodic job wait time estimates from a batch queue prediction service, user-specified intermediate nodes, and periodic network bandwidth measurements to deliver input data on time. Our evaluation shows as much as 91.0% reduction in staging times compared to direct transfers, 75.2% reduction in wait time on scratch, and 2.4% reduction in usage/hour. Thus, our JIT staging solution is able to reduce the duration of scratch space consumption and decrease the exposure window, while adapting to network volatility to deliver the data prior to job start, consequently improving HPC center serviceability.

Finally, we provide [75] a cloud storage framework for HPC, which utilizes proactive staging and offloading of data to cloud storage locations so as to have the input data available at the scratch storage — from multiple input sources — just before the job is about to run, and to offload output data — from scratch to the cloud — as soon as the job completes. The novelty of our approach lies in the transparent use of cloud storage resources as intermediate nodes, and bringing such resources to bear on the timely problem of HPC data delivery. Using this backdrop, we present several techniques to improve the end-user data delivery experience, by bringing data closer to the HPC center or the user so the data can be pulled eventually as needed to coincide job startup or a workflow deadline, respectively. CATCH exploits several desirable characteristics such as disseminating chunks to a geographically distributed set of locations, and extends them further, all in a seamless fashion to HPC users. Our results indicate that CATCH is able to: exploit orthogonal network bandwidth and adapt to network conditions, e.g., CATCH reduces average transfer times compared to direct transfers by as much as 81.1%; and mitigate the high cost of HPC I/O acquisition, especially for mid-size HPC workflows.

Together, these combined approaches create our Integrated End-User Data Service, wherein data transfer and placement on the scratch space are scheduled with job execution. This 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. Our strategies capture the current HPC usage pattern more accurately, and better equip the scratch storage system to serve the growing datasets of workloads.

7.2 Future Research Directions

In this dissertation, we have explored how to build advanced end-user data services that support HPC applications' growing data demands and help optimize scratch space usage. However, the modern distributed HPC workflow still poses many challenges to HPC system designers and application developers. A major challenge focused on by this dissertation is that data must be moved to and from an HPC center's scratch space frequently to accommodate the needs of end-users' workflows. In Chapter 6, we used cloud storage as an intermediate staging ground to hold a job's input and output data. A natural next step would be to extend our use of the cloud model and explore the demonstrated capabilities of cloud infrastructure in not just handling data storage, but also data-intensive applications and attempt to design high-end systems that support both scientific and enterprise workloads using cloud and HPC resources.

We are concerned with supporting applications that have both significant compute components, i.e., are HPC-friendly, and data processing components, i.e., are cloud-friendly. For example, pre-processing or post-processing, as well as smaller and simpler data manipulations can be performed using cloud resources, while the primary compute intensive part of

the workflow can still be shipped to the center and executed when necessary. This would result in more computation happening closer to data stored in the cloud, and thus reduces unnecessary data movement to and from the center, while making the scratch more available for the most compute intensive parts of HPC job workflows. Examples of such applications include space observation analysis, event detection in atomic colliders, and stock market modeling. In this context, a critical research challenge lies in determining and coordinating the right computation and storage resources to allocate to a job when using both HPC and cloud resources.

We intend to design and develop an integrated HPC and cloud platform to support emerging scientific and enterprise applications. This platform can potentially offer the HPC community the high-throughput data processing benefits of the cloud, while facilitating use of HPC resources as "accelerators" for the most compute intensive components of HPC and enterprise applications. This strategy enables quick and efficient utilization of cloud storage and compute resources for both the user and the HPC center. The overall goal is a holistic approach to deriving optimal performance from cloud and HPC resources by matching application tasks to the best resources.

Bibliography

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

[2] UC/ANL Teragrid Guide. http://www.uc.teragrid.org/tg-docs/user-guide.html/#disk, 2004.

[3] Planetlab: An open platform for developing, deploying and accessing planetary-scale services. http://www.planet-lab.org, 2005.

[4] Earth system grid. http://www.earthsystemgrid.org, 2006.

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

[6] The research and education data depot network (reddnet). http://www.lstore.org/pwiki/pmwiki.php?n=REDDnet.Infrastructure, 2007.

[7] Batch Queue Prediction. http://nws.cs.ucsb.edu/ewiki/nws.php?id=Batch+Queue+Prediction, 2008.

[8] DMOVER: Scheduled data transfer for distributed computational workflows. http://www.psc.edu/general/software/packages/dmover/, 2008.

[9] Internet2. http://www.internet2.edu/, 2008.

[10] Laser Interferometer Gravitational-Wave Observatory. http://www.ligo.caltech.edu/, 2008.

[11] National Institute of Computational Sciences. http://www.nics.tennessee.edu/computing-resources/kraken, 2008.

[12] National Lambda Rail: light the future. http://www.nlr.net/, 2008.

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

[14] Reddnet enabling data intensive science in the wide area. http://www.reddnet.org/mwiki/index.php/Main_Page, 2008.

[15] Spallation Neutron Source. http://www.sns.gov/, 2008.

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

[17] Sun constellation linux cluster. http://www.tacc.utexas.edu/resources/hpcsystems/#constellation, 2008.

[18] National Center for Computational Sciences. http://www.nccs.gov/, 2009.

[19] Nsf teragrid. http://www.teragrid.org, 2009.

[20] Amazon ec2 homepage. http://aws.amazon.com/ec2/, 2010.

[21] Bbcp. http://www.slac.stanford.edu/%7Eabh/bbcp/, 2010.

[22] FUSE: Filesystem in Userspace. http://fuse.sourceforge.net/, 2010.

[23] Gyrokinetic toroidal code (gtc) website. http://gk.ps.uci.edu/GTC/index.html, 2010.

[24] Nccs user support - data transfer. http://www.nccs.gov/user-support/general-support/data-transfer/, 2010.

[25] S3FS. http://code.google.com/p/s3fs/, 2010.

[26] Near Real Time Modeling of Weather, Air Pollution, and Health Outcome Indicators in New York City. http://cfpub.epa.gov/ncer_abstracts/index.cfm/fuseaction/display.abstractDetail/abstract/8649, 2012.

[27] H. Abbasi, M. Wolf, F. Zheng, G. Eisenhauer, S. Klasky, and K. Schwan. Scalable data staging services for petascale applications. In *Proc. ACM HPDC*, 2009.

[28] S. Albers and M. Büttner. Integrated prefetching and caching in single and parallel disk systems. In *Proc. 15th ACM SPAA*, 2003.

[29] W. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link. The Globus Striped GridFTP framework and server. In *Proc. ACM/IEEE SC*, 2005.

[30] S. Annapureddy, M. J. Freedman, and D. Mazires. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd USENIX NSDI*, 2005.

[31] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proc. ACM SIGMETRICS*, 2007.

[32] A. Bayucan, R. L. Henderson, C. Lesiak, B. Mann, T. Proett, and D. Tweten. Portable Batch System: External reference specification. `http://www-unix.mcs.anl.gov/openpbs/docs/v2_2_ers.pdf`, 1999.

[33] J. Bent, D. Thain, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Explicit control in a batch aware distributed file system. In *Proc. 1st USENIX NSDI*, 2004.

[34] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proc. 6th IOPADS*, 1999.

[35] V. Bhat, S. Klasky, S. Atchley, M. Beck, D. Mccune, and M. Parashar. High performance threaded data streaming for large scale simulations. In *Proc. 5th ACM/IEEE Grid*, 2004.

[36] A. R. Butt, T. A. Johnson, Y. Zheng, and Y. C. Hu. Kosha: A peer-to-peer enhancement for the network file system. *Journal of Grid Computing: Special issue on Global and Peer-to-Peer Computing*, 2006.

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

[38] L. Cherkasova and J. Lee. Fastreplica: Efficient large file distribution within content delivery networks. In *Proc. 4th USENIX USITS*, 2003.

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

[40] J. W. Cobb, A. Geist, J. A. Kohl, S. D. Miller, P. F. Peterson, G. G. Pike, M. A. Reuter, T. Swain, S. S. Vazhkudai, and N. N. Vijayakumar. The neutron science teragrid gateway: a teragrid science gateway to support the spallation neutron source: Research articles. *Concurrency and Computation : Practice and Experience*, 2007.

[41] B. Cohen. BitTorrent Protocol Specification, `http://www.bittorrent.org/protocol.html`, 2007.

[42] R. L. Collins and J. S. Plank. Downloading replicated, wide-area files – a framework and empirical evaluation. In *Proc. 3rd IEEE NCA*, 2004.

[43] Conseil Européen pour la Recherche Nucléaire (CERN). LHC– the large hadron collider, `http://lhc.web.cern.ch/lhc/`, 2007.

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

[45] Directed acyclic graph manager. `http://www.cs.wisc.edu/condor/dagman/`, 2007.

[46] Department of Energy, Office of Science. Innovative and Novel Computational Impact on Theory and Experiment (INCITE), http://www.er.doe.gov/ascr/incite/, 2008.

[47] A. Downey. Using queue time predictions for processor allocation. In *Proc. JSSPP*, 1997.

[48] Druschel et. al. Freepastry, http://freepastry.rice.edu/, 2004.

[49] M. Gleicher. HSI: Hierarchical storage interface for HPSS. http://www.hpss-collaboration.org/hpss/HSI/, 2008.

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

[51] Grid physics network: Data intensive science. http://www.griphyn.org, 2004.

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

[53] I. Iliadis, R. Haas, X.-Y. Hu, and E. Eleftheriou. Disk scrubbing versus intra-disk redundancy for high-reliability raid storage systems. In *Proc. ACM SIGMETRICS*, 2008.

[54] G. Khanna, U. Catalyurek, T. Kurc, R. Kettimuthu, P. Sadayappan, I. Foster, and J. Saltz. Using overlays for efficient data transfer over shared wide-area networks. In *Proc. ACM/IEEE SC*, 2008.

[55] S. Kiswany, M. Ripeanu, A. Iamnitchi, and S. Vazhkudai. Are peer-to-peer data dissemination techniques viable in todays data intensive scientific collaborations? In *Proc. 13th Euro-Par*, 2007.

[56] S. Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh. stdchk: A Checkpoint Storage System for Desktop Grid Computing. In *Proc. 28th IEEE ICDCS*, 2008.

[57] T. Kosar and M. Livny. Stork: Making data placement a first class citizen in the grid. In *Proc. 24th IEEE ICDCS*, 2004.

[58] D. Kostic, A. Rodriguez, J. Albrecht, A. Bhirud, and A. M. Vahdat. Using random subsets to build scalable network services. In *Proc. 4th USENIX USITS*, 2003.

[59] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. 19th ACM SOSP*, 2003.

[60] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proc. ACM SIGMETRICS*, 1999.

[61] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 2001.

[62] M. Li, S. Vazhkudai, A. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman. Functional partitioning to optimize end-to-end performance on many-core architectures. In *Proc. ACM/IEEE SC*, 2010.

[63] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proc. 8th IEEE ICDCS*, 1988.

[64] 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 *Proc. ACM ICS*, 2006.

[65] P. Maymounkov. Online Codes. Technical Report TR2003-883, 2002.

[66] N. Megiddo and D. S. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proc. 2nd USENIX FAST*, 2003.

[67] Microsoft. Windows Azure Platform, `http://www.microsoft.com/windowsazure/`, 2010.

[68] Microsoft. Windows Azure Pricing, `http://www.microsoft.com/windowsazure/pricing/`, 2010.

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

[70] H. Monti, A. R. Butt, and S. S. Vazhkudai. A result-data offloading service for hpc centers. In *Proc. ACM PDS Workshop at SC*, 2007.

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

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

[73] H. Monti, A. R. Butt, and S. S. Vazhkudai. /Scratch as a Cache: Rethinking HPC Center Scratch Storage. In *Proc. ACM ICS*, 2009.

[74] H. Monti, A. R. Butt, and S. S. Vazhkudai. Reconciling Scratch Space Consumption, Exposure, and Volatility to Achieve Timely Staging of Job Input Data. In *Proc. IPDPS*, 2010.

[75] H. Monti, A. R. Butt, and S. S. Vazhkudai. CATCH: A Cloud-based Adaptive Data Transfer Service for HPC. In *Proc. IPDPS*, 2011.

[76] H. M. Monti, A. R. Butt, and S. S. Vazhkudai. Timely result-data offloading for improved hpc center scratch provisioning and serviceability. *IEEE Transactions on Parallel and Distributed Systems*, 2011.

[77] H. M. Monti, A. R. Butt, and S. S. Vazhkudai. On timely staging of hpc job input data. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[78] National center for biotechnology information. `http://www.ncbi.nlm.nih.gov/`, 2005.

[79] K. Park and V. S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proc. 3rd USENIX NSDI*, 2006.

[80] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. 15th ACM SOSP*, 1995.

[81] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. ACM HotNets*, 2002.

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

[83] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Proc. Netstore*, 1999.

[84] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 1997.

[85] J. S. Plank. Erasure codes for storage applications. Tutorial Slides, presented at *USENIX FAST*, `http://www.cs.utk.edu/~plank/plank/papers/FAST-2005.html`, 2005.

[86] J. S. Plank, S. Atchley, Y. Ding, and M. Beck. Algorithms for high performance, wide-area distributed file downloads. *Parallel Processing Letters*, 2003.

[87] A. Riska and E. Riedel. Idle read after write: Iraw. In *Proc. USENIX ATC*, 2008.

[88] P. Rizk, C. Kiddle, and R. Simmonds. A gridftp overlay network service. In *Proc. 7th ACM/IEEE Grid*, 2007.

[89] P. Rodriguez, A. Kirpal, and E. W. Biersack. Parallel-access for mirror sites in the internet. In *Proc. IEEE Infocom*, 2000.

[90] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, 2001.

[91] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proc. 1st USENIX FAST*, 2002.

[92] B. Schroeder and G. A. Gibson. Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *Proc. USENIX FAST*, 2007.

[93] P. Schwan. Lustre: Building a File System for 1,000-node Clusters. In *Proc. Ottawa Linux Symposium*, 2003.

[94] Sloan digital sky survey. `http://www.sdss.org`, 2005.

[95] S. Shah and J. Elerath. Reliability analysis of disk drive failure mechanisms. In *Proc. RAMS*, 2005.

[96] R. Sherwood, R. Braud, and B. Bhattacharjee. Slurpie: A cooperative bulk data transfer protocol. In *Proc. IEEE Infocom*, 2004.

[97] W. Smith, V. Taylor, and I. Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In *Proc. JSSPP*, 1997.

[98] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proc. SIGCOMM*, 2001.

[99] W. Tang, N. Desai, D. Buettner, and Z. Lan. Analyzing and adjusting user runtime estimates to improve job scheduling on the blue gene/p. *In Proc. IPDPS*, 2010.

[100] D. Thain, S. S. J. Basney, and M. Livny. The kangaroo approach to data movement on the grid. In *Proc. ACM HPDC*, 2001.

[101] B. Tierney, J. Lee, M. Holding, J. Hylton, and F. Drake. A network-aware distributed storage cache for data intensive environments. In *Proc. ACM HPDC*, 1999.

[102] N. Tran and D. A. Reed. ARIMA time series modeling and forecasting for adaptive I/O prefetching. In *Proc. 15th ACM ICS*, 2001.

[103] S. Vazhkudai and J. Schopf. Predicting sporadic grid data transfers. In *Proc. 11th ACM HPDC*, 2002.

[104] S. Vazhkudai, J. Schopf, and I. Foster. Predicting the performance of wide-area data transfers. In *Proc. 16th IPDPS*, 2002.

[105] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proc. USENIX ATC*, 2004.

[106] W. X. Wang, Z. Lin, W. M. Tang, W. W. Lee, S. Ethier, J. L. V. Lewandowski, G. Rewoldt, T. S. Hahm, and J. Manickam. Global gyrokinetic particle simulation of turbulence and transport in realistic tokamak geometry. *Journal of Physics: Conference Series*, 2005.

[107] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computing Systems*, 1999.

[108] 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. ACM/IEEE SC*, 2007.

Appendix A

Simulating HPC Data Management with simHPC

The interplay between the different system components at an end-user site and the HPC center is complex and requires a controlled environment for in-depth analysis, which is near-impossible to do in real HPC setups. Thus, we have developed a realistic simulator to examine data management at HPC centers, *simHPC*, which models job execution, scratch management, and data movement.

A.1 Job Scheduling

In *simHPC*, jobs are scheduled using a First-Come First-Served (FCFS) policy with back-filling. Here, a number of large jobs are first scheduled in the order they arrive, until a majority of the machine's resources are allocated. Next, smaller jobs are scheduled. This approach is often employed in large-scale supercomputers, such as Jaguar [18], which are intended to run a few large jobs that take up most of the machine (e.g., 100,000 cores). However, such larger jobs can leave a small but significant number of cores idle; back-filling helps to avoid this by assigning smaller jobs to the idle cores. The goal is to strike a balance between the HPC center's desire to cater to "hero apps" that could take up an entire machine and potential idle cores.

A.2 Trace-Driven Simulation

simHPC utilizes a number of different traces to provide an accurate model of the system. Specifically it uses job and bandwidth traces.

Table A.1: Statistics about the job logs used by *simHPC*.

Duration	22753 Hrs
Number of jobs	80025
Job execution time	1 s to 120892 s, average 5849 s
Input data size	2.28 MB to 7481 GB, average 65.3 GB

A.2.1 Job Traces

The job traces were obtained from ORNL's Jaguar supercomputer [18] and represent nearly three years of job execution [73]. These traces provide for each job: arrival time, start time, total job execution time, and the compute resources used. Additionally, the traces also contain the amount of physical memory and virtual memory used by a job. The memory values and compute resources are used to estimate the amount of data used or created by a job, e.g., the product of the utilized memory per core and the number of cores requested by an application provides the size of a possible input, checkpoint, or output data item which needs to be moved to or from the scratch space [1]. Finally, each job in the trace corresponds to a job executing in our simulator. Table A.1 shows some relevant characteristics of the logs.

A.2.2 Bandwidth Traces

We model the intermediate nodes by using NWS bandwidth measurements from 50 different sites on the PlanetLab [81] testbed. The bandwidth traces provide pairwise bandwidth measurements for the 50 sites over a duration of 96 hours. Each simulated node in *simHPC* is assigned a measured trace. Since there are more nodes in the simulator than measured on PlanetLab some nodes will have duplicate bandwidth traces. Nodes running for longer than 96 hours simply loop through their associated trace. Since the measured bandwidth is for pair-wise exclusive communication, it does not capture the behavior when a node is participating in multiple transfers. In *simHPC*, we make an assumption that the available pair-wise bandwidth is reduced proportionally by the number of transfers in which a node is involved.

A.2.3 Simulator Output

simHPC provides an output trace with information about overall scratch space usage and the time it would take to transfer the required data to and from the scratch for a given job. This information can then further be used to determine any delay in meeting job scheduling deadlines.

[1]HPC centers neither log the submission scripts, nor the input and output data generated by specific jobs. It is not possible to change this behavior due to administrative reasons. Thus, we have to resort to such approximation.

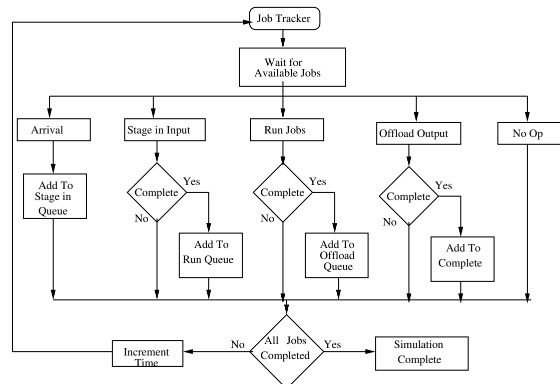


Figure A.1: Control flow in *simHPC*.

A.3 Flow of Control in *simHPC*

simHPC maintains a pool of nodes arranged in a configurable topology to use as intermediate nodes. Nodes are randomly selected to facilitate the simulated transfer. If a node is used for multiple transfers at the same time, the bandwidth is equally divided between the transfers. Moreover, *simHPC* can also capture varying storage capacities of the nodes and can alter transfer paths based on the capacities. In *simHPC*, we are mainly concerned with moving the data between the center and the first-level intermediate nodes only. Note that, while this can be easily extended to capture the end-user data delivery, we do not, as we can utilize our real implementation (Chapters 4, 5, and 6) to more accurately study such behavior.

Figure A.1 illustrates *simHPC*'s operation. The main driver is a *Job tracker* that reads the logs, and selects an appropriate action for the simulator to take. We have opted for using the same time-scale as the logs. At each job arrival, the tracker places it in a *wait queue*. The job input data staging is then started. The staging process may take many simulator ticks depending on the size of the input data, but once the process completes the job is moved to a *run queue*. The job will wait there until sufficient compute resources to run the job become available. Once the job completes its execution, it moves to the offload queue. If the simulator is modeling a decentralized offload, intermediate nodes will be chosen and the offload process will begin. If the standard approach is used, the data will remain on the scratch until it is purged by the center. Finally, *simHPC* also provides accounting and statistics about the staging and offloading process, such as the scratch space used and the data read, as well as other vital statistics.