CATCH: A Cloud-based Adaptive Data Transfer Service for HPC

Henry M. Monti, Ali R. Butt
Dept. of Computer Science
Virginia Tech
Email: {hmonti, butta}@cs.vt.edu

Sudharshan S. Vazhkudai
Computer Science and Mathematics Division
Oak Ridge National Laboratory
vazhkudaiss@ornl.gov

***Abstract*—Modern High Performance Computing (HPC) applications process very large amounts of data. A critical research challenge lies in transporting input data to the HPC center from a number of distributed sources, e.g., scientific experiments and web repositories, etc., and offloading the result data to geographically distributed, intermittently available end-users, often over under-provisioned connections. Such end-user data services are typically performed using point-to-point transfers that are designed for well-endowed sites and are unable to reconcile the center's resource usage and users' delivery deadlines, unable to adapt to changing dynamics in the end-to-end data path and are not fault-tolerant. To overcome these inefficiencies, decentralized HPC data services are emerging as viable alternatives. In this paper, we develop and enhance such distributed data services by designing CATCH, a Cloud-based Adaptive data Transfer serviCe for HPC. CATCH leverages a bevy of cloud storage resources to orchestrate a decentralized data transport with fail-over capabilities. Our results demonstrate that CATCH is a feasible approach, and can help improve the data transfer times at the HPC center by as much as 81.1% for typical HPC workloads.**

***Keywords*-HPC data management, data-staging, end-user data delivery, Cloud for HPC, HPC center serviceability**

I. Introduction

A. Problem Statement

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 [1], Large Hadron Collider [2]), collaborations (e.g., Earth System Grid [3]), state-of-the-art cyber-infrastructure (e.g., TeraGrid [4]) and supercomputers (e.g., Jaguar [5], Kraken [6]) 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 services* are often an afterthought in multi-million dollar HPC centers and cyber-infrastructure projects, leading to their sub-optimal use. *An elegant data delivery scheme can have a profound impact on user experience and also improve HPC center serviceability.* Extant, point-to-point data delivery techniques, commonly used in HPC centers, are unable to meet user delivery, job startup and scratch storage space purge deadlines, unable to adapt to changing dynamics in the end-to-end data path and are not fault-tolerant. Further, these transfer tools are only optimized for transfers between two already well-endowed sites [7]. In contrast, end-user data delivery involves providing access to the data at the user's desktop. It cannot be ignored as a "last-mile" issue. These inefficiencies can be prohibitive to sustaining high performance. In this paper, we aim to mitigate such lack of integrated data orchestration between HPC centers and end-users.

B. CATCH Overview

We present CATCH, a cloud-based adaptive data transfer service for HPC, that is able to seamlessly absorb the terabytes of data emanating from simulations or observations and move it closer to either the end-user or the HPC center. CATCH enables the offload of a large job's output data from an HPC center's parallel file system (PFS), *scratch space*, to cloud storage locations. To improve performance and avoid HPC center storage purge deadlines 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.

C. Contributions

CATCH provides a cloud storage framework for HPC, which utilizes proactive staging-in 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. Specifically, our contributions are:

Staged and decentralized offloading: We utilize a combination of both a staged as well as a decentralized delivery scheme for job data. This is a fundamentally different way of delivering job data in HPC centers and is a non-trivial endeavor. Compared to a direct transfer, our techniques have the added benefits of employing cloud storage nodes to provide resilience in the face of end-resource failure and the exploitation of available orthogonal bandwidth in the end-to-end data path.

Integration with Cloud Services: 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. 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 demonstrate ways in which these nodes can be specified and used within a scientific collaboration.

Integration with FUSE: We have exported our end-user data delivery service through the file system abstraction provided by FUSE [8]. End-user programs can thus write and read to cloud storage and move data through them using standard file system operations.

Integration with real-world tools: Our solutions are developed in the context of real-world tools that are commonly used in HPC, such as PBS [9] and NWS [10].

Detailed evaluation: Finally, we have evaluated CATCH using both a Windows Azure-based implementation and simulations driven by the Jaguar supercomputer job logs. Our approach optimizes precious scratch space usage and minimizes the exposure of input data at center storage. Results shows as much as 81.1% reduction in average transfer times under CATCH compared to direct transfers, and reduced exposure to scratch failures: 75.2% reduction in wait time on scratch, and 2.43% reduction in usage/hour.

II. Background and Enabling Technologies

In this section, we provide relevant background and describe the enabling technologies for CATCH.

A. A Case for End-User Data Services

We now describe the current methods for managing end-user data and their shortcomings, why end-user data services are critical to HPC center serviceability, and how the cloud can help facilitate such services.

Data Offloading: Result data from supercomputer jobs needs to be offloaded to end-users for local visualization or to another compute component of the distributed job workflow. This needs to be accomplished in a timely fashion both to meet a delivery constraint as well as to prevent the result data from getting purged from the HPC center scratch space that is typically reserved for currently running or soon to run jobs. The lack of 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.

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 Internet repositories, collaborating sites and other clusters that run pieces of the job workflow.

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. 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 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 usecases, 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, coupled with the observation that the cloud provides a number of distributed storage resources, naturally leads to the question of **how the cloud can be employed to mitigate the data delivery challenges in HPC**.

Previous Work: We have designed a framework [11], [12], [13] for 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 HPC center to/from end-user sites, thus addressing the issue of end-user site availability during point-to-point transfers. Our decentralized delivery

also factors in deadlines: i.e., for a timely data offload from the center or a timely staging to coincide with job-startup by synchronizing with center purge policies and job batch queue prediction services.

A significant drawback of our previous 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 a decentralized data offload and staging.

B. Using the Cloud for End-User Data Services

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.

Cloud as Intermediate Storage for Decentralized 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 [14]. Thus, a wide variety of HPC applications can be supported by the resulting data 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.

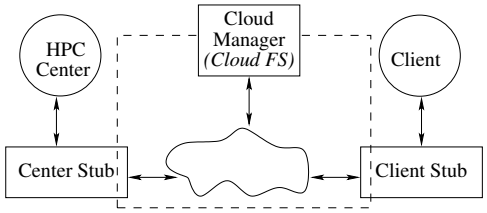


Figure 1. The main software components of CATCH.

Azure Data Services: We have used the Windows Azure platform [14] 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.

III. 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.

A. Design Overview

CATCH has three main software components as shown in Figure 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

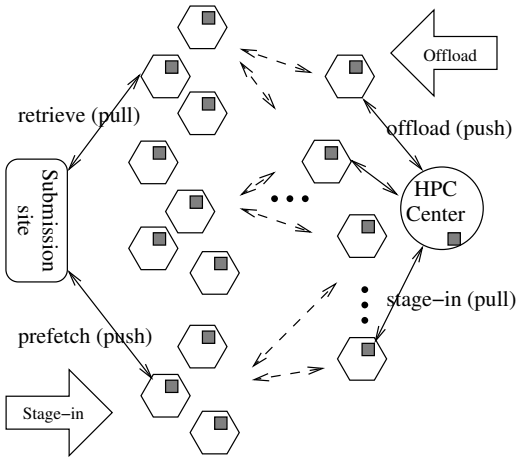


Figure 2. The data flow path from the HPC center to the end-user site. The intermediate resources are represented by hexagons. The gray squares represent software hooks/APIs that CATCH uses to control the data flow.

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 have 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 2 shows the high-level flow of data in CATCH.

We have developed detailed models for building end-user data services with ad hoc resources in our previous work [11], [13]. In CATCH, we overcome all of the issues arising from such ad hoc intermediate sites (discussed in Section II), 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.

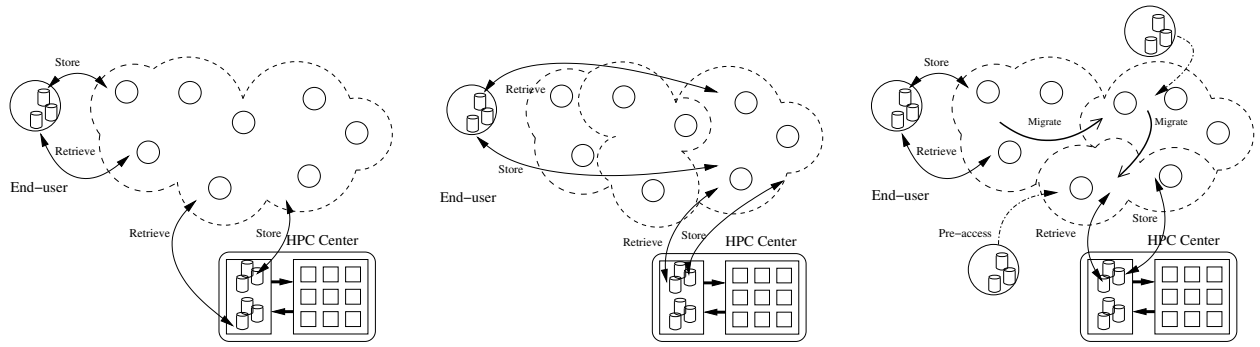
B. Cloud Data Interface

The cloud provides a suitable platform for developing and expanding end-user data delivery services. We have built our software using the Azure [14] platform. In the following, we discuss several possible heuristics for the HPC data transfer system to interact efficiently with the cloud.

1) 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 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, 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*.

2) 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 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.



(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 3. Different approaches for using the cloud to implement end-user data services.

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.

3) 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 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 [10], 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.

C. 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 [8] as *Cloud FS* (Figure 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 [15] for Amazon S3 or stdchk [16], [17], a file system atop distributed storage of disks, memory or SSD.) 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 IV. 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.

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.

```
#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 4. An example annotated job script.

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.

D. 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 [9]). Special directives can be used to annotate the job script as shown in Figure 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.

E. 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 [18], 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 jobs [13], 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

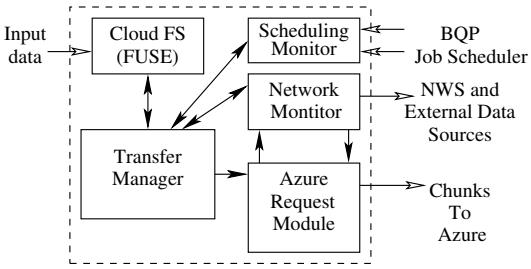


Figure 5. Architecture of CATCH.

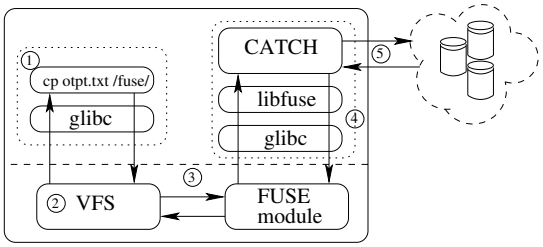


Figure 6. Architecture of FUSE-based *Cloud FS*.

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.

IV. Implementation

We have implemented CATCH using about 2500 lines of C# code with the Windows Azure [14] 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.

A. Architecture

The main components of CATCH are shown in Figure 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 *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) [19] 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 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. 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).

B. Data Operations

Access to the Windows Azure [14] 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.

1) 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.

2) 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.

3) 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 orches-

trate 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 for moving data from R_2 to R_1 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.

C. 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.

V. Evaluation

In this section, we present an evaluation of CATCH using both our implementation of Section IV and an analysis driven by three-year job-statistics logs from the Jaguar [5] supercomputer. We also compare our results to the popular direct transfer techniques that are the default approach for transferring data in many HPC centers.

A. 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.

1) 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 I. 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

Table I
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

Table II
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 III
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

the slowest region is in Asia. This is expected as this region is provided to be primarily accessed by people close to Asia.

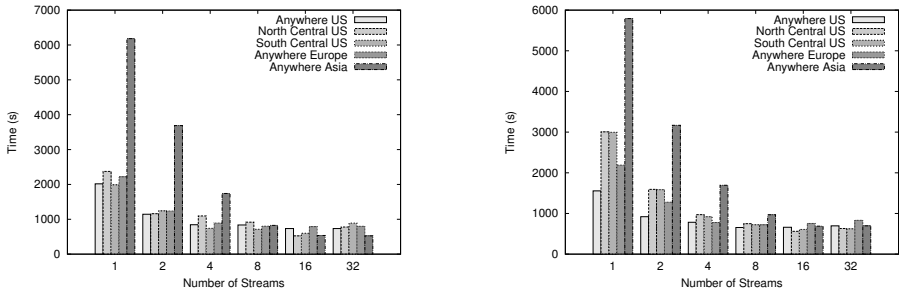
2) Effect of Multiple Transfer Streams on Access Times: During our previous experiments we observed that Azure is capable of handling many simultaneous requests, which can provide high aggregate throughput. To take advantage of this ability we designed CATCH to utilize multiple streams for transferring 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 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 number 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 offer 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.

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, i.e., the exposure window (E_w), consequently, improving center serviceability.

3) 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 II 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 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.

4) Comparison with Direct Transfer Methods: For our next experiment, we utilized our 5 cloud storage accounts coupled with 5 PlanetLab [21] 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 [11].

Info About Transfer Programs: We compared several point-to-point direct transfer tools that are prevalent in HPC: (i) scp, a baseline secure transfer protocol; (ii) IBP [22], 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 [7], an extension to the FTP protocol, which provides authentication, parallel transfers, and allows TCP buffer size tuning for high performance; and BBCP [23], which also provides high performance through parallel transfers and TCP buffer tun-



(a) Data staging times from end-user site (Write).

(b) Data retrieving times from HPC center (Read).

Figure 7. Transfer times (in seconds) to different cloud regions, using increasing numbers of streams. The file size used is 1 GB.

Table IV
STATISTICS ABOUT THE JOB LOGS USED IN THE SIMULATION STUDY.

Duration	22753 Hrs
Number of jobs	80025
Job execution time	30 s to 120892 s, average 5849 s
Input data size	2.28 MB to 7481 GB, average 65.3 GB

ing. Note that these protocols are all typically supported [24] by HPC centers such as Jaguar [5].

Table III 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 services.

B. Simulation

In this section, we study the performance of timely staging using job-statistics logs collected over a period of three-years on the Jaguar [5] supercomputer. Table IV shows some relevant characteristics of the logs.

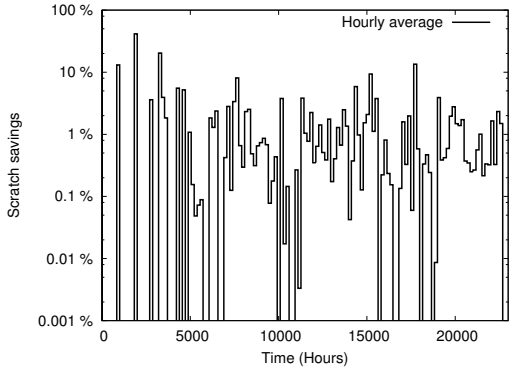


Figure 8. Scratch savings under CATCH compared to direct `scp`, calculated per hour. Purge period is seven days.

To analyze the logs, we have developed a simulator that captures the design of our setup. The simulator models job queuing, scheduling, batch-queue prediction, job execution times, and provides data about scratch space usage and delay in meeting deadlines. It also models distributed intermediate nodes, their bandwidth variations and decentralized data staging. It uses the connectivity values from the cloud and plays the periodic snapshot of bandwidth measurements to emulate volatility. In the following, we use this simulator to gain insights into end-user data services.

1) Impact on Scratch Space Usage: In this experiment, we quantify the impact of decentralized transfers on scratch space usage. We play the logs in our simulator and determine the amount of scratch used both under `scp`-based direct transfer and CATCH. For this test, we assume that the scratch is empty at the beginning, and has perfect information about when the data will be used by the job, i.e., job start up time is known. 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 8 shows the average savings per hour in scratch space usage by CATCH. It can be seen that CATCH is able to provide up to 45.3% in scratch savings.

Table V
CURRENT AZURE PRICING.

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

Table VI
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

We also calculated the average savings per hour across the entire log, and found that CATCH 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, CATCH offers a viable means for conserving precious scratch resource.

2) Effect on Exposure Window: In the next 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. We found that for 30.7% of the jobs, CATCH was effectively able to reduce E_w to zero, and for the remaining jobs it reduced E_w by 64.2%, i.e., 75.2% reduction on average across all jobs. Moreover, E_w was reduced by at least a factor of 10 for 48.3% of the jobs. Overall, the significantly reduced E_w for most jobs under CATCH shows that it can provide better resiliency against storage system failures and costly re-staging.

C. Cost of Cloud Usage

In the next experiment, we determine how the cost of cloud services impact CATCH usage. Table V shows the current pricing structure used by Azure [18]. Table VI 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.

VI. RELATED WORK

The use of intermediate buffers to hide latency is used in Kangaroo [25] for Grid computing, with the goal to provide reliability against 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.

The GridFTP overlay network service [26], [27] implements a specialized data storage interface (DSI) to achieve split-TCP functionality. IBP [22] offers a data distribution infrastructure with a set of strategically placed resources (storage depots) to move data, and implement what is referred to as logistical networking. Our approach of providing a file system view of the data transport is similar to IBP's logistical networking and exNodes. The main difference between these approaches and ours is that instead of relying on specialized resources, we leverage general-purpose cloud resources to achieve end-user data delivery.

Stork [28], a scheduler for data placement activities in a grid environment, is used to schedule data and computation together. However, these systems still use point-to-point transfer tools. Consequently, these solutions cannot address network volatility either.

A number of systems such as Bullet[29], [30], Shark [31], CoDeeN [32], and CoBlitz [33] 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. HPC end-user delivery requires factoring in center-user service agreements and dynamic cloud resource availability, which are not considered in these systems. Our work shares with these systems the goal of utilizing multiple paths for transferring large amount of data, but differs in its focus on HPC applications and use of cloud nodes for intermediate storage.

VII. CONCLUSION

In this paper, 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 the timely problem of HPC data delivery. To this end, CATCH provides a FUSE-based file system abstraction to the decentralized data flow through the cloud storage substrate. 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 tools are integrated with real-world HPC tools such as PBS, batch-queue prediction, and several popular HPC point-to-point tools. 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%; reduce exposure to scratch failures, e.g., 75.2% reduction in wait time on scratch; and mitigate the high cost of HPC I/O acquisition, especially for mid-size HPC workflows.

ACKNOWLEDGMENTS

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 and CNS-1016793.

REFERENCES

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

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

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

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

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

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

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

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

[9] Albeaus Bayucan, Robert L. Henderson, Casimir Lesiak, Bhroam Mann, Tom Proett, and Dave Tweten. Portable Batch System: External reference specification. http://www-unix.mcs.anl.gov/openpbs/docs/v2_2_ers.pdf, 1999.

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

[11] Henry Monti, Ali R. Butt, and Sudharshan S. Vazhkudai. Timely offloading of result-data in hpc centers. In *Proc. ICS*, 2008.

[12] Henry M. Monti, Ali R. Butt, and Sudharshan S. Vazhkudai. /scratch as a cache: rethinking hpc center scratch storage. In *Proc. ICS*, 2009.

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

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

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

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

[17] M. Li, S.S. Vazhkudai, A.R. 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. Supercomputing*, 2010.

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

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

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

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

[22] 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.

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

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

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

[26] P. Rizk, C. Kiddle, and R. Simmonds. A gridftp overlay network service. In *Proc. International Conference on Grid Computing*, 2007.

[27] 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. Supercomputing*, 2008.

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

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

[30] Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin M. Vahdat. Using random subsets to build scalable network services. In *Proc. USITS*, 2003.

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

[32] 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.

[33] KyoungSoo Park and Vivek S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proc. NSDI*, 2006.