Design of an Active Storage Cluster File System for DAG Workflows

Patrick Donnelly
Department of Computer Science and Engineering
University of Notre Dame
384 Fitzpatrick Hall
Notre Dame, IN 46556
pdonnel3@nd.edu

Douglas Thain
Department of Computer Science and Engineering
University of Notre Dame
384 Fitzpatrick Hall
Notre Dame, IN 46556
dthain@nd.edu

ABSTRACT

We present the conceptual design of Confuga, a cluster file system designed to meet the needs of DAG-structured workflows. Today's premier cluster file system Hadoop is commonly used to support large peta-scale data sets on commodity hardware and to exploit active storage through Map-Reduce, a specific workflow pattern. Unfortunately, DAG-structured workflows have very different requirements from Map-Reduce workflows: whole-file access is standard and multiple dependencies are common. Confuga will meet these new requirements by replicating rather than striping files as in Hadoop, by exploiting DAG-structured workflow consistency semantics, and by permitting multiple dependencies in job descriptions. To the end user, Confuga will appear as a drop-in replacement for a batch system and a file system, combined into a single entity that can be invoked by existing workflow managers. In this paper, we describe the design philosophy of Confuga, sketch the major components of the system, and explain how the system will behave under expected workloads.

Keywords

Workflow, Data-Intensive, Active Storage, Data-Locality, Hadoop, Confuga, Chirp

1. INTRODUCTION

The last decade has seen rise of new scalable cluster file systems used for storing scientific data and for enabling executing tasks on this data. In fact, creating large storage clusters using commodity hardware has never been easier. Hadoop [33] is perhaps the most popular open source implementation of a scalable cluster file system and is used across industry and academia for scaling to peta-byte datasets. Hadoop is designed to enable the Map-Reduce [7] workflow abstraction to allow processing on these immobile datasets and has led an entire paradigm of research on exploiting data-local execution.

DISCS-2013, November 18 2013, Denver, CO, USA

ACM 978-1-4503-2506-6/13/11 . . . $15.00.
http://dx.doi.org/10.1145/2534645.2534656

Concurrently, distributed and parallel scientific computing applications have given rise to systems designed to execute thousands to millions of tasks, including Condor [15] and Sun Grid Engine [10]. These distributed compute engines are designed with a centralized job submission site which deploys tasks and input data to execute nodes, providing a simple interface for distributed computation. There has been extensive work to express scientific workflows in formats suitable for faster and automated deployment on these systems. Usually, this involves distilling the workflow into a set of tasks and input/output file dependencies to form a directed acyclic graph (DAG). Distributed systems including Dryad [13], DAGMan [9], and Makeflow [3] express workflows in this way. Following generation, the DAG is executed by submitting tasks to a distributed computing engine with output files gathered at each step for the next task in the DAG.

Unfortunately, DAG-structured workflows exhibit poor performance on Hadoop. Primarily, this is because Hadoop optimizes for the Map-Reduce workflow abstraction which is incompatible with generic DAG-structured workflows. DAG-structured workflows are generally written where tasks consume entire input files. On Hadoop, blocks are replicated and striped across the cluster, so any running task will need to perform remote I/O to fetch blocks not resident on the storage node. The design of the job submission platform is also tailored for Map-Reduce jobs composed of hundreds to thousands of tasks, each performing a Map operation on a file block; partially as a result of this design around large jobs, Hadoop's job bandwidth suffers for single-task jobs used by DAG-structured workflows [3]. Finally, data locality in Hadoop is achieved only in the context of Map-Reduce: a job has a single input file with a Map task executed for each of the input blocks striped across the cluster. A job's tasks cannot have multiple dependencies while achieving complete data-locality.

To answer the need for data-intensive DAG-structured workflows, we have designed **Confuga**[1], an active storage cluster file system. Confuga replicates data with file-granularity striped across the cluster, executes unmodified task applications within a POSIX file system context, and offers multi-dependency data-local execution. Confuga is optimized for DAG-structured workflows where coarser consistency semantics allow for fewer metadata operations. This means all I/O operations are scoped at the whole-task level: input file access is performed before a task begins and global visibility of

[1]Confuga is a portmanteau of the Latin word *con* and Italian's *fuga* (also Latin). A fugue (fuga) is a contrapuntal musical composition composed of multiple voices. Here the intended meaning is "with many voices" although it literally translates as "with chase". [Later it was determined confuga is itself a Latin word meaning "refugee".]

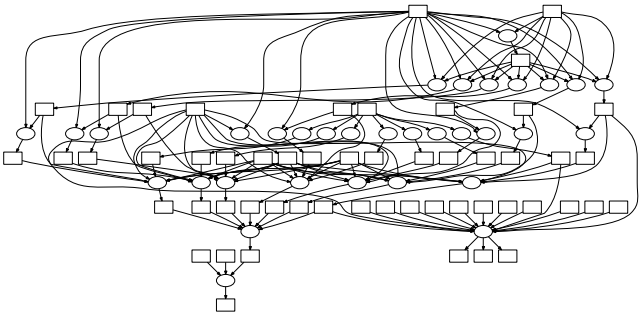

Figure 1: A typical DAG-structured Workflow. This comes from a bioinformatics workflow using SHRiMP [23]. Here files are boxes, circles are tasks, and lines indicate dependencies.

output file manipulations are committed after the task completes. These semantics enable a number of optimizations that Confuga will exploit to achieve performance.

Confuga is currently a work-in-progress. It will be used as a drop-in replacement for a batch system and a file system, combined into a single entity that can be invoked by existing workflow managers.

2. OBSERVATIONS ON DAG-STRUCTURED WORKFLOWS

Scientific workflows often re-use large datasets in multiple workflows. It is expensive to dispatch a large set of data to multiple execution nodes for each workflow execution. It is beneficial to persistently store data in a cluster where the work of replicating the data to multiple execution nodes is done once.

Metadata interactions occur at task start/end. DAG-structured workflows are designed to have known data dependencies and data outputs for each task. This allows the scheduler to order task execution and, for some systems, schedule subsequent tasks near recently produced dependencies.

DAG-structured workflows are written for whole-file access. Workflows are written so that tasks consume entire files. Usually this requires splitting larger files before hand and adjusting the application but in practice this is not a significant burden on users. However, it is also common for some scientific workflows to have tasks which require large input files that cannot be split. This creates difficulties in distribution and storage for the workflow manager.

3. CONFUGA

Confuga is an active storage cluster file system designed for executing DAG-structured workflows. At a high-level, Confuga is composed of a head-node and multiple storage nodes. The head-node presents to users a regular global namespace for storing files as a regular directory hierarchy. Files are striped across the cluster, with multiple replicas. The head-node also allows for workflow managers to submit jobs to the cluster. These jobs are single-task regular applications with explicit file access descriptions. Applications access/write data as regular files on the local file system but visibility of changes are only committed to the global namespace at task completion.

Figures 2-4 show operations of interest on the file system. S_i is a storage node holding files (F_i). HN is the head node composed of a replica manager (RM), namespace manager (NM), and job

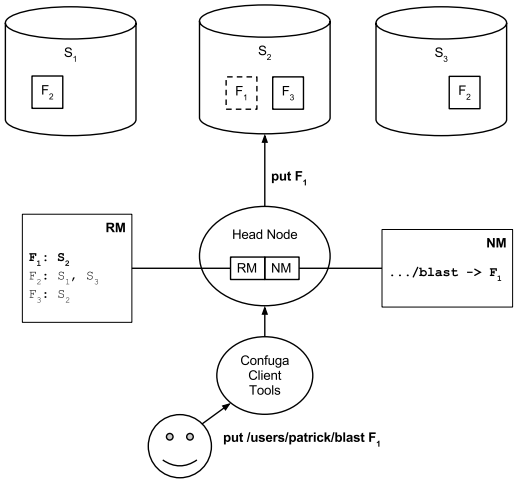

Figure 2: Put

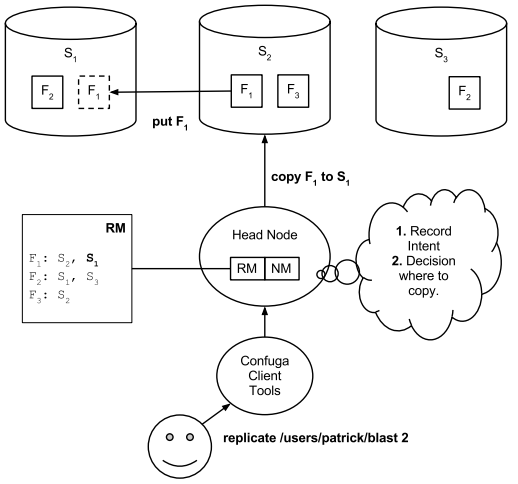

Figure 3: Copy

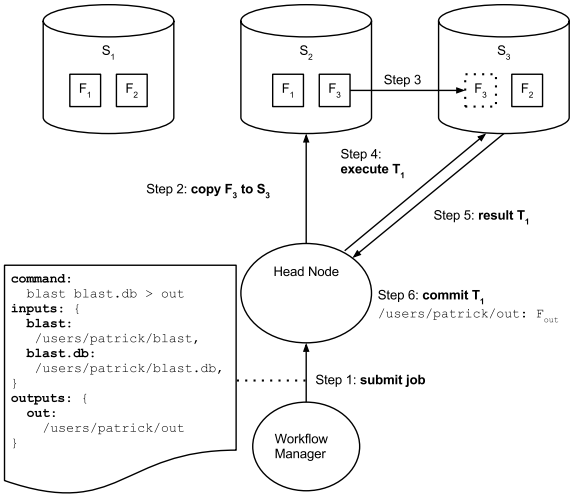

Figure 4: Execute

scheduler which dispatches tasks (T_i) to storage nodes for execution.

Figures 2 and 3 show a user doing manipulations of the file system using command-line tools. Figure 2 illustrates a scenario where a user might upload a file which may be replicated numerous times in anticipation of heavy use. Figure 3 displays the user issuing an intent to the file system for a file to be replicated. For this case, Confuga records the intent and repairs the file system over time by replicating the file as needed. Figure 4 shows the steps the cluster takes to execute a job submitted by a workflow manager. We will examine the details of these components and operations in the following sections.

3.1 Properties

Replicated data at file-granularity: Replicated storage has obvious benefits for data redundancy in case of failure and has become a requirement for modern cluster file systems. Replicas also serve as a caching mechanism where the cluster adjusts replication of files based on demand. Maintaining the cache state for each storage node enables the scheduler to make smarter data-locality decisions for tasks and balance load across the cluster.

The granularity of replicas will be individual files. Replication of smaller blocks has some attractive benefits such as deduplication of similar files and large file support (because blocks of a large file can be stored across storage nodes); however, block granularity increases work for whole-file access typical in DAG-structured workflows.

(Restricted) POSIX I/O Interface: Most existing workflows are composed of applications that utilize POSIX I/O. This necessitates that execution of an unmodified task executable occur within a POSIX-compliant file system. In practice this means that tasks must execute on a system with a kernel driver that mounts the distributed file system, with FUSE [26] access to mount the file system in userspace, or with a system call interposition agent which redirects local I/O to remote services, such as Parrot [27]. This, however, has impractical system access constraints (root/fuse access) or performance problems (interposing system calls: reduced performance).

Instead, Confuga will create a binding in the local file system (either through links or file copies) to the replicas which exist on the storage node[2]. This requires that Confuga know what exact files a task will access: the major constraint of job submission in Confuga. Each job must come with binding information for the task's namespace to the global namespace. This allows for tasks to have a regular POSIX interface to its files but prevents manipulation of the global namespace or dynamic access to files in the global namespace. This also means tasks (processes) cannot intercommunicate except through the file system where changes are made visible globally at task completion, and, only tasks which start after those changes are made visible may see those changes.

Consistency maintained at task-boundaries: Most file systems respect POSIX consistency semantics by making changes visible to readers immediately, `write` for example. In practice, this restricts performance of a distributed file system. AFS [12] is well-known for dispensing with `commit-on-write` consistency semantics in favor of `commit-on-close`. This means changes to a file are only visible to readers once the file is closed by the writer. Confuga takes this a step farther by exploiting the properties of DAG-structured workflows by restricting visibility of changes to task completion. We refer to this as `commit-on-exit`. These semantics ultimately reduce load on the metadata server (head node).

Likewise for `read`, partially because we know which files a task will use prior to execution, we can limit its visibility of changes to a file to the beginning of its execution. This restriction is attractive since it prevents metadata checks for file changes and the subsequent file delta retrieval and enforces DAG-structured workflow semantics.

Global and Task Namespace Management: As with most distributed file systems, there will be a shared global namespace accessible to all users of the system. This global namespace will facilitate sharing data with other users for their workflows.

Each file in the global namespace references an identifier or inode. The inode is derived from the file's contents as a cryptographic hash value. This allows storage nodes to map stored replicas to a flat namespace and to generate a universally unique identifier[3] for a replica without consulting the metadata server. This is a common technique in file systems known as content addressable storage.

Confuga's management of task namespaces will be a distinguishing feature of the system. Specifying a job will require knowing all input files and output files for the task. Additionally, task execution is considered atomic and its side-effects on the global namespace are only committed after execution completes. A task's description sent to a storage node will include a mapping of the task namespace to file identifiers, which index replicas. Output files generated by a task are put into the global namespace by the head-node before the job is considered finished.

3.2 Architecture

The system will be composed of four main components: a job scheduler, a replica manager, a namespace manager, and multiple storage nodes. A head node will be responsible for maintaining the job scheduler, replica manager, and namespace manager[4]. All file system metadata operations are performed through communication with the head node.

Replica Manager The replica manager is responsible for maintaining a list of storage nodes which have a copy of a file. A file is looked up by using a file identifier (inode). File identifiers can be derived universally as the cryptographic hash of the file contents.

In addition to storing replica locations, the replica manager stores miscellaneous metadata for files including the size and creation date and maintains certain cluster upkeep. This would include periodically checking that storage nodes have correct replicas and that sufficient replicas exist to ensure redundancy.

Namespace Manager The namespace manager maintains the directory hierarchy of the file system. Directories and symbolic links are maintained in a mirror file system layout on the head node's local file system. Regular files are references to a file identifier.

The namespace manager is also responsible for access control. An access control list is maintained for each directory. The mechanism will enforce read, write, list, insert, delete, and administration rights for individual users or groups.

Job Scheduler The Confuga file system will support a number of job creation and manipulation commands which will be passed to the job scheduler. The scheduler is responsible for creating a static task description which includes a task namespace. This task

[2]File copies are expensive even on the local disk but ensure tasks cannot modify original replicas with direct access via links. More advanced local file systems such as ZFS [4] that support snapshots or light-weight file system clones (copy-on-write) would provide better mechanisms for cheap and safe binding of replicas to a task namespace.

[3]Ignoring infinitesimally small chances of collision.

[4]Failure of the head node is an important thing to avoid and design around. Redundant head nodes would solve this, and Confuga could be made to support this, but we avoid adding the complexity for now.

namespace binds input files to file identifiers and output files to the global namespace (to be committed upon task completion). As part of the result of a finished task, the storage node includes the file identifiers of each output to be mapped into the global namespace.

The job scheduler queues jobs waiting to be run and determines where and when jobs are executed on storage nodes. It is required that all input data be resident on a storage node before the job can be executed there. The scheduler is also capable of replicating data to a new storage node to increase the number of concurrent jobs. Alternatively, it can hold a job until a storage node with the job's dependencies becomes available.

Job execution is considered *atomic*. A job reads all input files from the global namespace in one operation. A job's task is executed on a storage node and its outputs committed as one operation to the global namespace. If any part of committing output fails, then the entire commit is discarded and the job is considered failed without side-effects. As an optional job failure semantic, a job description could require that input files are unchanged before committing output files to the global namespace.

3.3 Load Balancing

Confuga inherently balances load of the cluster in a number of respects:

Metadata Server (Head Node) Load As part of adapting the design of the cluster to handle DAG-structured workflows, Confuga can perform all metadata operations for a task before dispatch. This is both because the scheduler knows what files the task will require as part of the job description and because the consistency semantics depart from POSIX by preventing visibility of changes to the global namespace during task execution.

This philosophy for reducing head-node load is not uncommon. For example, Hadoop's Map-Reduce implementation knows which blocks a task will work on. It also knows these blocks cannot change due to the write-once nature of Hadoop's FS.

Another mechanism for reducing load on the metadata server is the use of content addressable storage. This allows the allocation of an inode number for each file without communication with the metadata server. Also, consistency of file contents is simply maintained. Old replicas become unreferenced when the namespace manager updates a path but not unusable. This allows tasks currently using those replicas to continue functioning normally. There is no need to synchronize the storage cluster nodes when a filename in the global namespace refereinces a new identifier.

Storage Nodes Load Because Confuga replicates files across the cluster, each node with a replica increases the amount of parallelism we can achieve. We use whole file replicas because this is compatible with DAG-structured workflows.

In contrast, block striping of files across the cluster is used by Hadoop to maximize parallelism of tasks. Hadoop is able to do this because Map-Reduce splits an input file, assigning a task to each split. For Confuga, a DAG is whole-file based. Splitting a file is actually harmful since a (generic) task needs to fetch the entire file during its execution. A user may instead split a large file into smaller files. This can be done before executing any workflows or as part of the DAG. Either strategy is common practice by our collaborators.

3.4 Implementation

Confuga is currently a work-in-progress. Covering the implementation in detail is not possible due to space considerations but a brief mention of pertinent details follows.

Confuga leverages an existing deployable file system service, Chirp, to handle remote I/O access, authentication, and access control. Developing Confuga requires clustering a set of Chirp servers to form a coherent global namespace, file location layout, and job submission platform.

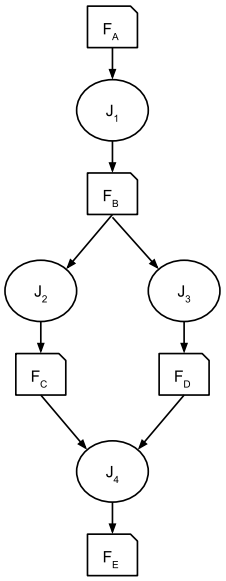

Figure 5: Example Workflow This is a simple DAG workflow with trivial fan-out and fan-in behavior. As before, F_i is a file and J_i is a job. We examine the details of evaluating this workflow in Section 4.

Chirp [28] is a file system service designed to be deployed on a grid to make available data for remote execution sites. Functionally and conceptually, Chirp is similar to NFS [24]. Chirp is attractive for deployment on grids and clusters for a number of reasons including trivial user deployment without administrator privileges, strong and usable authentication and access control mechanisms, and multiple mechanisms for applications access.

Part of Confuga's design allows storage nodes to be completely unaware of the larger context, the cluster file system. Storage nodes can run regular Chirp servers. Replicas are maintained on storage nodes as regular files managed by the head node. Jobs are submitted to storage nodes with task namespace binding information devoid of the global namespace references. Each task namespace binds a name in the task context, e.g. `app.exe`, to a file identifier known to be on the storage node, e.g. `/store/eabd1234...`. [Recall that file identifiers are simply file names derived by the hash of the file contents.]

The head node also operates a Chirp server as an interface to the cluster. Normally each remote procedure call (RPC) to Chirp, e.g. `chmod`, results in an equivalent system call to the underlying file system. For Confuga, the head node instead operates with an overlaid back-end file system which interprets each RPC in the context of Confuga. Most metadata operations like `chmod` are interpreted regularly and reflected in the global namespace which is simply mirrored on the local file system. Other operations such as putting or getting a file would result in the name node redirecting the operation to a storage node. A job submission RPC would result in a scheduler making a decision about which storage node to send the job to, based on the location of replicas of dependencies.

4. EXAMPLE WORKFLOW

To illustrate how Confuga operates, we will walk through a sim-

ple workflow shown in Figure 5. It is composed of 4 jobs with typical fan-in and fan-out behavior. J_1 produces an output file which both J_2 and J_3 consume. J_4 consumes the output of J_2 and J_3.

The first step in evaluating the workflow will require the workflow manager (WM) to upload any input file dependencies, in this case F_A. This would also include any executable files that will be run, which we ignore for simplicity in this example. The Head Node (HN) streams uploaded files to storage nodes:

$WM->HN:$ UPLOAD F_A
$HN->S_1:$ PUT F_A

At this point, the WM can begin submitting its first job to the cluster. [Figure 4 illustrates submitting a job to the cluster.] The job description includes the command to execute, the input and output files, and other miscellaneous requirements. The HN dispatches the job for execution to S_1 with all input file dependencies resident.

$WM->HN:$ SUBMIT J_1
$HN->S_1:$ EXEC J_1
$WM->HN:$ WAIT J_1

Once J_1 is complete and its results committed to the global namespace, the WM may dispatch the next jobs in batch:

$WM->HN:$ SUBMIT J_2
$WM->HN:$ SUBMIT J_3
$HN->S_1:$ COPY F_B S_2
$S_1->S_2:$ PUT F_B
$HN->S_1:$ EXEC J_2
$HN->S_2:$ EXEC J_3
$WM->HN:$ WAIT J_2, J_3

Here J_2 and J_3 both require input file F_B. In this example, the HN determines that copying the file to another node prior to execution will have better performance (allowing two nodes to execute simultaneously). Once F_B is copied from S_1 to S_2, the HN dispatches J_2 and J_3 to the respective nodes.

The workflow ends with the WM sending J_4 to the HN. Since the two required input files are on two different nodes, the HN initiates a copy of F_D to S_1 from S_2. Finally, J_4 is sent to S_1 for execution.

$WM->HN:$ SUBMIT J_4
$HN->S_2:$ COPY F_D S_1
$HN->S_1:$ EXEC J_4
$WM->HN:$ WAIT J_4

Note: This example does not delve into the details of heuristics and algorithms the HN uses when making decisions on when and where to replicate files for executing jobs. The HN may decide, for fairness perhaps, to serialize execution of J_2 and J_3 on S_1. Evaluating these decisions will be subject of planned future research.

Expectations: We expect Confuga to perform well in situations where file size increases and thus cost to move files grows. In particular, when large input files are used regularly in workflows and are replicated across the cluster. As expected, transfers between nodes will also operate between storage nodes without the HN mediating the connection.

Confuga will perform less optimally for workflows which rely on partial reads of files, such as Map-Reduce creating many "Map" tasks each reading from a few blocks of a large file. This is because Confuga will require that the entire file be resident on the execution node before running the task. This follows from designing around the observation that DAG-structured workflows do whole-file reads for each task.

5. RELATED WORK

Distributed file systems like NFS [24] and AFS [12] are often used in multi-user systems with reasonably strict POSIX compliance. AFS is notable for introducing write-through-on-close, where POSIX consistency semantics are relaxed so other clients will not see changes to a file until the writer closes the file. Other POSIX extensions for high-performance computing have been proposed [32] which allow batching metadata operations and explicitly relaxing certain consistency semantics.

Still, all parallel cluster file systems suffer from a metadata bottleneck [2, 6], even for highly performant cluster file systems such as IBM's GPFS [25] and PVFS [22]. For Confuga, metadata issues have been largely avoided by designing the system for workflow semantics, where file access is known at dispatch and visibility of changes are only committed on task completion. This allows Confuga to batch many operations (`open`, `close`, `stat`, and `readdir`) at task dispatch and completion and opportunistically prohibit dynamic file system access.

Content-addressable storage (CAS) is a common feature in file systems allowing for simple universal inode generation (the inode is the checksum), deduplication, and indexing files through a flat namespace. Many distributed file systems use CAS including Venti [19] and HydraFS [30]. Both Venti and HydraFS use block granularity for storing objects to achieve better deduplication, especially important in an archival system. Git [29] and Mercurial [17] are distributed version control systems which use CAS to uniquely identify commits and objects where remote repositories are infrequently available.

Active Storage [1, 14, 21, 20] was originally proposed as a technology where computation be moved to the hard disk drives. This would free up the CPU for other tasks and eliminate the need to move data into main memory through congested buses. For many reasons, including lack of storage manufactor interest, this idea has not been implemented.

Object based storage devices (OSD) [11] provides some support for using the processing power of disks. An OSD provides an abstracted storage container which produces objects tagged with metadata. A set of operations for manipulating and searching/selecting the objects on the devices is part of the standard [31]. OSD has become an integral component of numerous file systems including Lustre [5] and Panasas [16]. Computation on Lustre storage servers has also been supported [8] to allow client programs to have direct access to data, and in [18] as a user-space solution.

Other work on Active Storage has centered around the use of workflow abstractions with workflow managers tightly coupled to the storage system. Map-Reduce [7] is an abstraction used in Hadoop [33] for executing tasks near data, preferring a storage node storing the desired block, a storage node on the same rack, or anywhere in the cluster. The abstraction is very specific and only allows expressing data locality needs in terms of a single file.

6. CONCLUSIONS

This paper discusses the design of Confuga, a cluster file system for executing DAG-structured workflows. Confuga is currently a work-in-progress. We are confident it will perform well for data-intensive workflows our collaborators have. Future work will evaluate the behavior of the system with different scheduling algorithms, replication strategies, and workflows.

7. REFERENCES

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *ACM SIGPLAN Notices*, volume 33, pages 81–91. ACM, 1998.

[2] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni. Parallel i/o and the metadata wall. In *Proceedings of the sixth workshop on Parallel Data Storage*, pages 13–18. ACM, 2011.

[3] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 1. ACM, 2012.

[4] J. Bonwick and B. Moore. ZFS: The last word in file systems, 2007.

[5] P. J. Braam and R. Zahir. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.

[6] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig. Small-file access in parallel file systems. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11. IEEE, 2009.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] E. J. Felix, K. Fox, K. Regimbal, and J. Nieplocha. Active storage processing in a parallel file system. In *Proc. of the 6th LCI International Conference on Linux Clusters: The HPC Revolution*, 2006.

[9] J. Frey. Condor DAGMan: Handling inter-job dependencies. *University of Wisconsin, Dept. of Computer Science, Tech. Rep*, 2002.

[10] W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35–36. IEEE, 2001.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, et al. File server scaling with network-attached secure disks. *ACM SIGMETRICS Performance Evaluation Review*, 25(1):272–284, 1997.

[12] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[14] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *ACM SIGMOD Record*, 27:42–52, 1998.

[15] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor-a hunter of idle workstations. In *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111. IEEE, 1988.

[16] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53. IEEE Computer Society, 2004.

[17] B. O'Sullivan. Distributed revision control with Mercurial. *Mercurial project*, 2007.

[18] J. Piernas, J. Nieplocha, and E. J. Felix. Evaluation of active storage strategies for the lustre parallel file system. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 28. ACM, 2007.

[19] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST*, volume 2, pages 89–101, 2002.

[20] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, 2001.

[21] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, pages 62–73. Citeseer, 1998.

[22] R. B. Ross, R. Thakur, et al. Pvfs: A parallel file system for linux clusters. In *in Proceedings of the 4th Annual Linux Showcase and Conference*, pages 391–430, 2000.

[23] S. M. Rumble, P. Lacroute, A. V. Dalca, M. Fiume, A. Sidow, and M. Brudno. SHRiMP: accurate mapping of short color-space reads. *PLoS computational biology*, 5(5):e1000386, 2009.

[24] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985.

[25] F. B. Schmuck and R. L. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, page 19, 2002.

[26] M. Szeredi. Fuse: Filesystem in userspace. 2005. *URL http://fuse.sourceforge.net*.

[27] D. Thain and M. Livny. Parrot: An application environment for data-intensive computing. *Scalable Computing: Practice and Experience*, 6(3):9–18, 2005.

[28] D. Thain, C. Moretti, and J. Hemmes. Chirp: a practical global filesystem for cluster and grid computing. *Journal of Grid Computing*, 7(1):51–72, 2009.

[29] L. Torvalds and J. Hamano. Git: Fast version control system. *URL http://git-scm. com*, 2010.

[30] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *FAST*, pages 225–238, 2010.

[31] R. O. Weber. Information technology-SCSI object-based storage device commands (OSD). *Technical Council Proposal Document T*, 10:92, 2004.

[32] B. Welch. POSIX IO extensions for HPC. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[33] T. White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.