Confuga: Scalable Data Intensive Computing for POSIX Workflows

Patrick Donnelly, Nicholas Hazekamp, Douglas Thain
Department of Computer Science and Engineering
University of Notre Dame
{pdonnel3,nhazekam,dthain}@nd.edu

***Abstract*—Today's big-data analysis systems achieve performance and scalability by requiring end users to embrace a novel programming model. This approach is highly effective whose the objective is to compute relatively simple functions on colossal amounts of data, but it is not a good match for a scientific computing environment which depends on complex applications written for the conventional POSIX environment. To address this gap, we introduce Confuga, a scalable data-intensive computing system that is largely compatible with the POSIX environment. Confuga brings together the workflow model of scientific computing with the storage architecture of other big data systems. Confuga accepts large workflows of standard POSIX applications arranged into graphs, and then executes them in a cluster, exploiting both parallelism and data-locality. By making use of the workload structure, Confuga is able to avoid the long-standing problems of metadata scalability and load instability found in many large scale computing and storage systems. We show that Confuga's approach to load control offers improvements of up to 228% in cluster network utilization and 23% reductions in workflow execution time.**

I. Introduction

Today's big-data analysis systems achieve performance and scalability by requiring end users to embrace a novel programming model. For example, Map-Reduce [1], Spark [2], and Pregel [3], all require restructuring the application workflow to match a restricted framework to achieve scale. This approach is highly effective whose the objective is to compute relatively simple functions on colossal amounts of data. The small expense of writing or porting a small, widely known algorithm (such as k-means clustering) to these new platforms is well worth the payoff of running at colossal scale.

However, in other realms of computing, porting to a new system is not so celebrated. For example, in scientific computing, high value programs with very sophisticated algorithms are typically developed in a workstation environment over the course of many years and validated through painstaking experiment and evaluation. Rewriting these codes for new environments is neither attractive nor feasible. As a result, the scientific computing community has embraced a *workflow* model of computing whereby standard sequential applications are chained together into large program graphs with a high degree of parallelism. These workflows are typically run on conventional clusters and grids rather than data-intensive computing systems.

To address this gap, we introduce Confuga, a scalable data intensive computing system that is largely compatible with the standard POSIX computing environment. Confuga brings together the *workflow* model of scientific computing with the *storage architecture* of other big data systems. End users see Confuga simply as a large file system tree that can be mounted in the ordinary way. Confuga accepts standard POSIX applications and runs them within the cluster, taking data locality into account. Large parallel workloads are expressed in the scientific workflow style, and can be as simple as writing a conventional Makefile.

Previous large scale storage systems have suffered from two problems as both systems and workloads scale up. (1) *Metadata scalability.* Any file system that provides a global namespace must have a global service to provide metadata regarding the location and status of each file. This service can be implemented as either a centralized server or as a distributed agreement algorithm, but either way, the service does not scale. Confuga avoids this problem by exploiting the structural information available in the workload. (2) *Load instability.* As workloads scale up, and the number of simultaneous users increase, it is all too easy for concurrent transfers and tasks to degrade each other's performance to the point where the entire system suffers non-linear slowdowns, or even outright task failures. For example, it is frequently observed that too many users running on Hadoop simultaneously will cause mutual failures [4]. Confuga avoids this problem by tracking the load placed on the system by each task or transfer, and performing appropriate load management, resulting in a stable system.

In this paper, we present the design objectives and overall architecture of Confuga. As a running example, we use three large bioinformatics workloads based on existing applications, BLAST and BWA, all running on a 24-node storage cluster. Through these examples, we demonstrate that Confuga minimizes global metadata access, provides data locality, enables efficient re-execution of similar tasks, and provides stability under high load conditions. We show that Confuga's approach to load control offers improvements of up to 228% in cluster network utilization and 23% reductions in workflow execution time.

II. Confuga – An Active Storage Cluster FS

Confuga[1] is an active storage [5] cluster file system designed for executing DAG-structured workflows. Confuga is composed of a single head node and multiple storage nodes.

[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".]

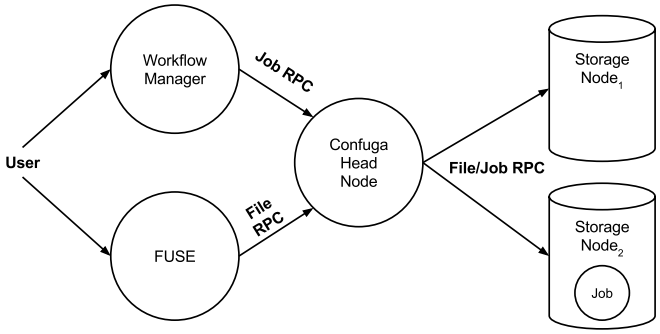

Fig. 1. Confuga Architecture

The head node manages the global namespace, indexes file locations and metadata, and schedules jobs on storage nodes. The individual storage nodes run as dumb independent active storage servers. The architecture is visualized in Figure 1.

A. Use as a File System

Confuga can be used as a regular distributed file system, with the global namespace and metadata centrally managed and with file replicas distributed across the cluster on storage nodes. Files in Confuga are organized in a usual hierarchical directory structure. The file system may be operated on using using a POSIX-style RPC protocol served by the Confuga head node daemon.

Users may interact with Confuga using FUSE [6] mounts or using its API, allowing trivial integration with their existing tools or regular UNIX utilities. Confuga has only a few caveats for external file system use: file writes are globally visible only after closing the open file handle and files may only be written once. Confuga also includes several useful functions that facilitate a collaborative file system environment for users. Confuga supports several authentication mechanisms including Kerberos and Globus and provides per-directory access control lists allowing fine-grained sharing of data with other users.

B. Use as Active Storage

While Confuga can be used a normal file system, its primary purpose is executing jobs close to their data. During normal use, a client will upload a dataset which will be acted on by the client's workflow. Alternatively, the client might reuse previously uploaded data or data shared by a collaborator on the same cluster. Once the dataset is resident on Confuga, the user may execute jobs through a *submit* and *wait* interface with explicit job dependencies and outputs. Typically, the user will do this using a high-level workflow manager which handles execution details like workflow-level fault tolerance, job ordering and dependency management.

Each job in Confuga specifies an opaque POSIX executable to run as a regular process without privileges. Confuga does not require changing the application code or hooking or modifying libraries for execution. Jobs execute within a private sandbox in a local POSIX file system, constructed from the job's explicitly defined namespace and local system resources. Figure 2 shows an example JSON [7] job specification.

```
{
 "executable":"./script",
 "arguments":["./script","-a"],
 "files":[
  {
   "confuga_path":"/u/joe/bin/script",
   "sandbox_path":"./script",
   "type":"INPUT"
  },{
   "confuga_path":"/u/joe/data/db",
   "sandbox_path":"data/db",
   "type":"INPUT"
  },{
   "confuga_path":"/u/joe/workflow/out1",
   "sandbox_path":"out",
   "type":"OUTPUT"
  }
 ]
}
```

Fig. 2. A simple job's description for Confuga. This program executes `./script` with the arguments `"./script" "-a"`. The first argument is the UNIX `argv[0]` for the program name. Confuga pushes the files `/u/joe/bin/script` and `/u/joe/data/db` from the global namespace into the job sandbox as `script` and `data/db`, respectively. When the job completes, the `out` file is pulled into the global namespace. These job descriptions are typically generated programatically by the workflow manager.

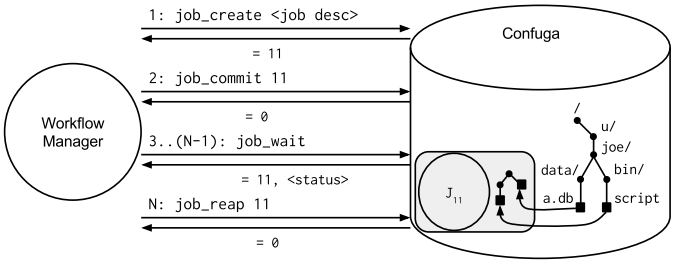

Fig. 3. Confuga Job Execution Protocol

The job's namespace maps all input file dependencies (including the executable) from the global Confuga file system namespace into the job's namespace. File names and directory structures may map to paths completely different from the global namespace. The `sandbox_path` provides a relative path for the mapping into the job's sandbox. The `confuga_path` refers to the file in the global namespace. Files are read from the global namespace prior to job execution. Output files are written into the global namespace after job completion.

The Confuga head node offers a job RPC protocol for clients to create jobs. Figure 3 shows how a job is dispatched and reaped by a client. Each operation uses two-phase commit, to prevent loss of acquired remote resources due to faults (e.g. a job is created and run but the client never receives the job identifier).

C. DAG-structured Active Storage Data Model

The fundamental idea in Confuga is the structure and semantics of a job, which impacts the system design and all opportunities for optimization. A job is executed as an atomic

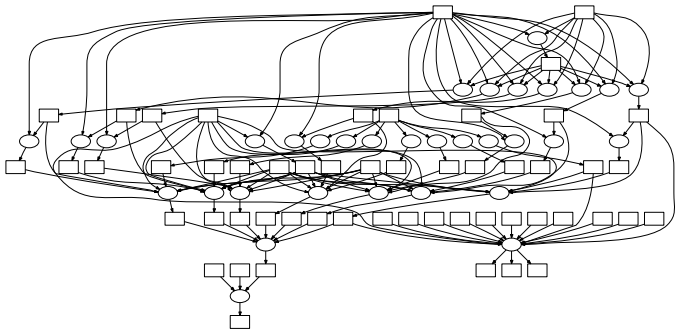

Fig. 4. A typical DAG-structured Workflow. This comes from a bioinformatics workflow using SHRiMP [8]. Here files are boxes, circles are tasks, and lines indicate dependencies.

code that does not interact with other concurrent jobs in any way. Its interaction with the cluster file system is defined in terms compatible with a DAG-structured workflow: inputs are pushed from the workflow namespace into the job sandbox prior to execution; each output file in the job sandbox is pulled into the workflow namespace on completion.

What this amounts to is a layering of consistency semantics. At the level of a job, it executes within a namespace with normal POSIX consistency semantics. All operations within its namespace are native accesses; the cluster file system does not introduce any remote file operations or new mount points. At the level of the workflow, consistency is maintained at job boundaries, start and end. We refer to these semantics as `read-on-exec` and `write-on-exit`. Restricting consistency semantics in this way has a number of advantages:

(1) Because each job includes its full list of data dependencies, Confuga is able to make smarter decisions about where, and most importantly when, to schedule a job to achieve complete data-locality and to control load on the network.

(2) Since all input files can only be pulled from the workflow namespace before the job begins execution, Confuga can eliminate dynamic/remote file operations by a job. This is an essential aspect of Confuga's execution model as it allows for controlling load on the network: a job cannot dynamically open files on the cluster which would introduce network overhead out of the head node's control.

(3) We are able to reduce metadata load on the head node by isolating storage nodes from the global cluster namespace. Any metadata required for a job and its files can be sent as part of job dispatch and retrieved when reaping the job. In fact, storage nodes in Confuga are treated as dumb active storage file systems, completely unaware of the global context.

Restricting a system to a certain execution model usually has drawbacks; Confuga is no different in this regard. While the model is flexible enough to run any type of workflow expressed as a DAG of jobs, it will not perform optimally when the input files are largely unused. Either because there are input files which are never opened by the job or because a large input file is not fully utilized. Confuga also requires that all dependencies are fully replicated on the storage node chosen for a job's execution. This means that all of a job's files must fit on disk. This requirement encourages structuring a large data set as multiple files, which is already a de facto requirement for DAG workflows. Our experience suggests this is not a significant burden on users.

D. Plugging a Workflow Manager into Confuga

Confuga only considers one job at a time for scheduling, without knowledge of the complete workflow. A workflow manager is responsible for the details of handling workflow fault-tolerance, execution statistics, task ordering and dependencies. For this purpose, we use **Makeflow** [9], which is a DAG-structured workflow manager using the venerable Make syntax for expressing the dependencies between tasks. The order of execution is determined by constructing a directed acyclic graph of the entire workflow, derived from the Makeflow specification file.

Supporting Confuga required adapting Makeflow to interface with the Confuga file system. This involved configuring the workflow manager to set a root, or current directory, for the workflow namespace within the Confuga namespace (e.g. `/u/joe/workflow/`) and using remote file system operations for checking dependency existence and successful output creation. Finally, a new job execution module was introduced for creating jobs using Confuga's protocol. These changes to Makeflow were straightforward, requiring a patch of less than 1000 lines of code.

III. Implementation

A. Confuga Head Node

The Confuga head node has several responsibilities including maintaining the global namespace, indexing file replicas, performing necessary replication, and scheduling jobs on storage nodes. The head node embodies most of the complexity and distinguishing aspects of Confuga. It is composed of three major components: a namespace (and metadata) manager, a replica manager, and a job scheduler.

The namespace manager stores the global namespace on a locally mounted regular POSIX file system. Confuga offers an API for most POSIX I/O namespace and metadata operations. These operations, such as `stat`, `link` or `symlink`, are executed literally within the local representation of the global namespace. Regular files receive the only special treatment in the namespace. A unique replica identifier is stored within each regular file. This identifier is used to index replicas of a file's content. Confuga includes an API for atomic `lookup` and `update` of a file's replica identifier.

The replica manager maintains a flat namespace of replicas indexed using a GUID (globally unique identifier) or using the hash of the replica (content address storage). For strong hash algorithms (like the one Confuga uses), the hash is also a GUID. Using a hash of the replica allows for easy deduplication, saving future replication work and decreasing storage use. The choice of using a hash or a GUID depends on the cost of generating the hash. New replicas created by jobs are assigned a random GUID when the replica is very large, this is discussed further in Section IV. Replicas are striped across the cluster as whole files. This allows storage nodes to cheaply map a replica into the job namespace using regular UNIX operations (`link` or `symlink`).

B. Confuga Scheduler

The Confuga scheduler drives all job execution on the cluster. It uses database transactions for all job operations reflected within a job state machine. The first task for a new job is to recursively `lookup` all input files and directories from the Confuga namespace, creating an equivalent mapping for directories and files to the replica namespace. Once a job has all of its inputs bound, the scheduler may proceed to schedule a job on an available storage node for execution. Currently, Confuga only allows one job to execute on a storage node at a time. Scheduled jobs are not immediately dispatched, first missing dependencies are replicated on the storage node by the scheduler.

Once a storage node has all of a job's dependencies, the scheduler may dispatch the job. The scheduler uses the composable job protocol discussed in Section II-B for dispatching jobs and reliably tracking their state within a database. The job protocol allows for the trivial remapping of the Confuga namespace (specified by the workflow manager) to the replica namespace, which is the only visible namespace at storage nodes. When a job completes at a storage node, the job result includes the replica identifier for each output file. The final task for a completing job is to perform an `update` for all output files, setting the replica identifier for files in the Confuga namespace.

C. Technologies Used

Confuga is implemented on top of an existing distributed file system, Chirp [10]. Starting a Confuga cluster is as simple as executing a Chirp server configured to be the Confuga head node and normal Chirp servers on each individual storage node. All of these services may be started without any privileges, allowing regular users to quickly erect a cluster file system for temporary or extended use. The Chirp server acting as the Confuga head node performs several functions including securely and reliably serving file operations on the Confuga file system over the network, overlaying access control for all file operations, and leveraging existing tools for seamless application access. Users may interact with Confuga using the Chirp library or command-line toolset, FUSE [6], or Parrot [11].

The head node also uses a SQLite database for managing the state of the cluster: location of file replicas, storage node heartbeats and the state of jobs. Jobs and replication are evaluated as state machines (a) to allow recovery in the event of faults by the head node or by the storage nodes and (b) to make scheduling and replication decisions using the complete picture of the cluster's current activities.

D. Cluster Hardware

For this work, we use a rack of 26 Dell PowerEdge R510 servers running RedHat Enterprise Linux 6.6, Linux kernel 2.6.32. Each server has dual Intel(R) Xeon(R) CPU E5620 @ 2.40GHz, for 8 cores total, 32GB DDR3 1333MHz memory, a 1 gigabit link, and 12 2TB hard disk drives in a JBOD configuration. Confuga's storage nodes each use a single disk formatted with the Linux *ext4* file system.

E. Benchmark Workflows

We have used two unmodified bioinformatics workflows, BLAST and BWA, for benchmarking Confuga. The BLAST workflow is composed of 24 jobs with a shared 8.5GB database. It is used for comparing a genomic sequence query against a reference database, yielding similar sequences meeting some threshold. The BWA workflow performs a genomic alignment between a reference genome and a set of query genomes. The BWA workflow is composed of 1432 jobs, starting with a 274 way split of the 32GB query genome. The purpose of this alignment is to later compare how well the genomes align and where in the reference genome they align.

IV. Metadata Load

Distributed file systems providing a global namespace for file metadata and location have always struggled with designing for extreme scale. POSIX consistency semantics in a distributed setting have significantly contributed to this problem. AFS [12] is well known for relaxing certain requirements to meet scale and performance needs. In particular, AFS introduced *write-on-close* so other clients will not see changes to a file until the writer closes the file. Doing otherwise would require propagating the update to the file's blocks and size to the file server before any write may complete.

Confuga minimizes load on the head node by exploiting the structural information available in the workflow. Specifically, Confuga relies on the complete description of the input and output files available from the DAG workflow manager. Using this information, Confuga is able to perform all metadata operations, including access control checks and determining replica locations, prior to job dispatch to a storage node. The job description sent to the storage node contains a complete task namespace mapping to immutable replicas local to the storage node. The storage node requires no further information to execute the job. Figure 5 visualizes the different models for handling metadata operations for Confuga and traditional POSIX distributed file systems.

It is worth noting that using workflow information in this way to optimize metadata operations is not uncommon in cluster file systems. For example, because Hadoop's Map-Reduce implementation knows which blocks a Map task will work on, it can minimize future work by isolating the Map task from the file system by looking up file metadata and replicas for the Mapper and forwarding data blocks directly to the Map function. Naturally, not following the Map-Reduce model results in performance penalties, e.g. by opening numerous other data files in HDFS. Additionally, Hadoop also relies on HDFS's immutable file blocks to reduce (eliminate) consistency concerns.

Confuga also relies on the jobs operating atomically, without consistency updates from the head node. This is a common constraint in DAG-structured workflow managers that synchronize and order tasks through output files. When jobs would run on a master/worker framework or on a cycle scavenging campus grid, it was not useful to allow network communication between jobs because (a) jobs would need to be able to "find" other jobs, (b) communication must get through firewalls between subnets, and (c) communication would introduce dependencies between jobs unexpressed in

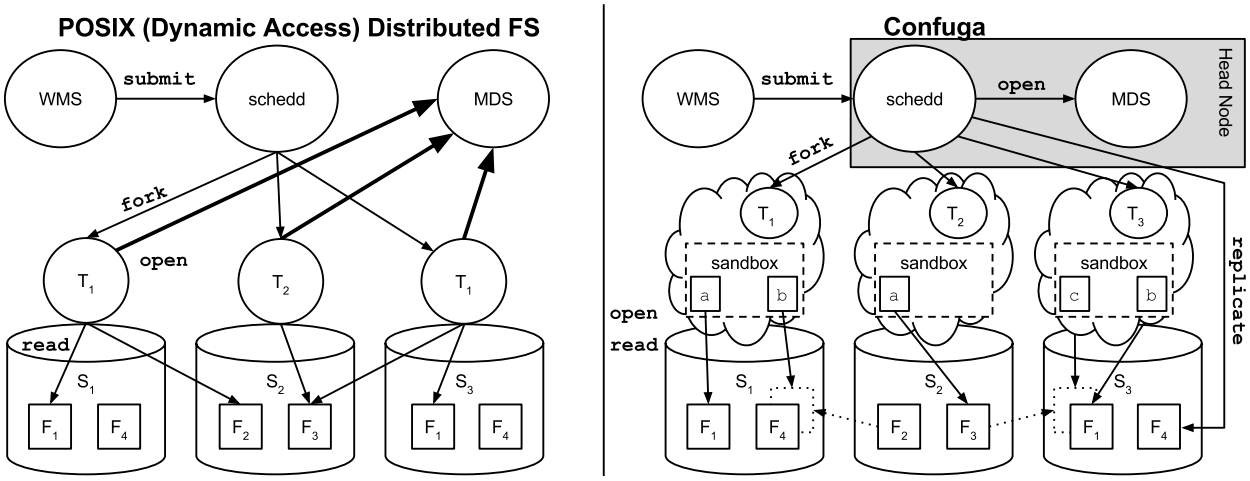

Fig. 5. Distributed File System Metadata Access Patterns. [Large cardinality as bold arrows.]

TABLE I. METADATA AND HEAD NODE OPERATIONS FOR BLAST WORKFLOW (24 JOBS)

Scheduler → MDS		**Scheduler → Storage Node**				**Storage Node → File System**			
Metadata	Count	Job	Count	Transfers	Count	All	Count	Sandbox	Count
lookup	881	job_wait	440679	thirdput	618	stat	14085	stat	4687
readdir	779	job_create	30	rename	597	open	3648	open	1407
update	104	job_commit	26	access	597	access	436	readlink	361
opendir	19	job_reap	24	-	-	readlink	400	getcwd	247
-	-	job_kill	0	-	-	getcwd	247	getdents	14
-	-	-	-	-	-	getdents	14	-	-
-	-	-	-	-	-	statfs	7	-	-

TABLE II. METADATA AND HEAD NODE OPERATIONS FOR BWA WORKFLOW (1432 JOBS)

Scheduler → MDS		**Scheduler → Storage Node**				**Storage Node → File System**			
Metadata	Count	Job	Count	Transfers	Count	All	Count	Sandbox	Count
lookup	13170	job_wait	90672	access	2350	open	17928	open	10731
update	3578	job_create	1433	thirdput	1452	access	1432	stat	1
readdir	0	job_reap	1432	rename	1448	stat	21	-	-
opendir	0	job_commit	1432	-	-	readlink	2	-	-
-	-	job_kill	0	-	-	-	-	-	-

Head and Storage Node Operations on the cluster during for the BLAST and BWA workflows. The Confuga scheduler performs file replica lookups and recursive directory reads with the metadata server for each job. This is a one-time operation for each job to bind files. Likewise, when a job finishes, the scheduler updates the global namespace with each job output file. For job and transfer operations in both workflows, there were several instances where the scheduler needed to repeat an operation due to lost connections.

We recorded the system calls for jobs run on the storage nodes using the `strace` utility. The tables show the file system operations done within the entire storage node namespace including local system files, libraries, and executables ("All") and operations executed within the job's namespace ("Sandbox").

the workflow description. Confuga takes advantage of the DAG workflow model by eliminating consistency checks and dynamic file access for jobs.

Finally, Confuga utilizes Content Addressable Storage (CAS) and Globally Unique Identifiers (GUID) for allocating a replica (filename) for a job's output file. Which technique is used depends on the size of the file. The goal of CAS is to deduplicate small files (by default defined as 16MB) on the cluster. When an output file is so large that generating the checksum becomes prohibitively expensive, a random GUID is used instead. While using a GUID is an optimization to allow a job to complete quickly, Confuga can generate the checksum for deduplication in the future. The use of CAS/GUID allows storage nodes to generate the file name (location) of the replica without the head node's assistance.

We evaluated metadata operations performed by Confuga and its jobs for two bioinformatics workflows, BLAST and BWA, shown in Tables I and II. Note that because the metadata server (MDS) and scheduler are both part of the head node, all of these metadata operations are locally handled.

There was an average of 74 BLAST (24 jobs) and 12 BWA (1432 jobs) metadata operations (`lookup`, `readdir`, `update`, and `opendir`) performed by the scheduler. Within each job's sandbox, there were an average of 293 (BLAST) and 7 (BWA) operations (`stat`, `open`, `readlink`, and `getdents`) that would have required interaction with the metadata server. For BWA, the ratio of `lookup+update` to `open+stat` is not as favorable as BLAST because each job would dynamically open up only some of its input files. These tests demonstrate that Confuga is able to significantly reduce traffic on the head node by batching metadata operations before

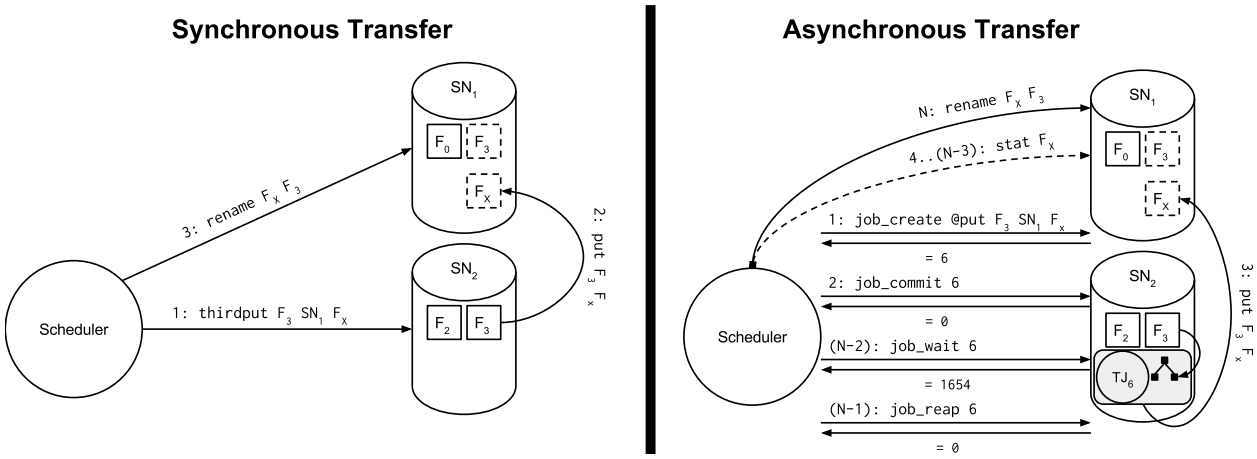

Fig. 6. Confuga Transfer Modes Synchronous transfers are blocking replication operations performed by the scheduler. The `thirdput` RPC is used to have a storage node transfer its replica to another storage node. Asynchronous transfers are a series of short blocking operations that create a **transfer job** on the storage node. The transfer job executes asynchronously with the head node. Operations are numbered in order.

and after a job.

V. Returning to the Data

Researchers typically work on the same data repeatedly, making minor tweaks to the workflow. As expected in an active storage system, Confuga allows running a workflow multiple times without repeatedly moving data to the execution site. So, the first workflow pays the price of distributing the data and subsequent workflows may execute more quickly.

Figure 7 shows two variations of "returning to the data". Each graph indicates the activity on all storage nodes, both for transfers in/out and for jobs executing. For these tests, we have configured Confuga to use a simpler synchronous transfer mechanism when moving replicas between storage nodes. This means that the scheduler does a blocking transfer for each replica, preventing any other work during the transfer. We talk about synchronous transfers more in Section VI-A.

In the first graph, we execute the same BLAST workflow twice consecutively. An 8GB dataset is uploaded immediately prior to running the first workflow, with replicas striped randomly across the cluster. The BLAST workflow executes with 24 long running jobs that each share an 8GB dataset split into multiple input files. The goal is to have the second run of BLAST benefit from the prior replication work during the first run. From the graph, one can see that Confuga is initially busy pulling the files from multiple storage nodes prior to executing the first job on SN_{12}. In order to increase concurrency, Confuga tries to replicate common input files to other storage nodes, arbitrarily choosing storage nodes 1 and 12 as sources. The final stage of the BLAST workflow is 3 fan-in jobs which gather previous outputs, causing several transfers at `t=01:45`. Because the two BLAST workflows are identical and the outputs are small, these results were deduplicated so the second execution of BLAST runs the 3 analysis jobs without any fan-in transfers.

In the second graph, we run the same BLAST workflow twice consecutively, as before, but also run the BWA workflow concurrently with the first run of BLAST. The 1432 job BWA workflow begins with an initial fan-out job that splits a 32GB dataset into 274 pieces. For the duration of the BWA workflow, these splits are transferred from SN_{11} to 6 other storage nodes. These transfers appear continuous and concurrent only because of the graph's minimum width of a transfer is 30 seconds (to ease visibility). The final fan-in job is also run on SN_{11} which gather outputs from the other nodes. The intent of this experiment is to determine how Confuga responds to two workflows with disjoint datasets. We can see that the BWA workflow taxed the scheduler with the large number of jobs and synchronous transfers, preventing complete use of the cluster. Eventually, SN_8, SN_9, and SN_3 (briefly) were picked up by BWA late in the run. Because SN_8 was claimed by BWA, SN_6 was chosen for the next BLAST job (with some dependencies already there).

These results tell us that Confuga is able to help existing unmodified workflows benefit from active storage. We see that the second run of BLAST runs about 19% faster. Additionally, running a another workflow concurrently with a disjoint dataset does not significantly impact the data-locality for jobs.

VI. Scheduling and Load Control

This section examines the consequences of allowing asynchronous and concurrent replication within the cluster. The goal of concurrent replication is to fully utilize the available network resources. As usual, concurrency is not a completely positive change. The cluster may not be able to fully use the resources it allocates or the accounting overhead of concurrency may slow down the head node.

For this work, Confuga uses a simple First-In-First-Out (FIFO) scheduler for all configurations. There has been significant effort in the community for constructing schedulers emphasizing fairness [4] (for users), smarter placement, and rescheduling [13]. For this section, we are focusing on how to control load on the cluster, particularly the network **after** scheduling (placing) jobs.

A. Synchronous and Asynchronous Replication

The Confuga scheduler uses two replication strategies for transferring job dependencies through the cluster: *synchronous*

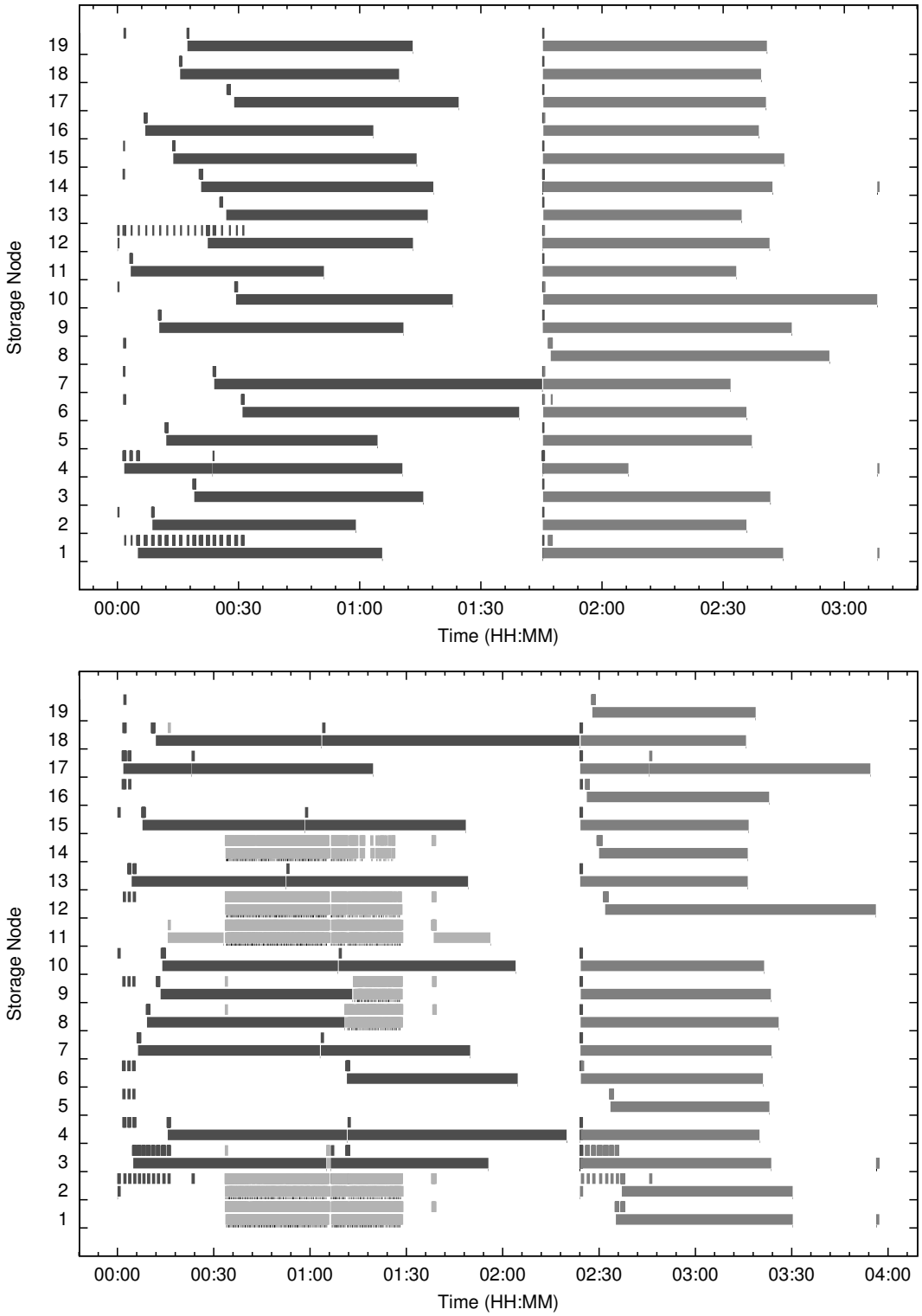

Fig. 7. These graphs visualize the activity at each storage node during a run of workflows. Each storage node row has two bars for activity: the top bar (small tics) shows transfers in or out and the bottom bar (long tics) shows job execution. The width of the bar indicates the duration of the activity. Each transfer or job also has a minimum width of 30 seconds, to ease visibility. Additionally, dots below the job execution bar show when a job has finished to distinguish consecutive jobs.

The top graph has two sequential runs of the same BLAST workflow, each run colored differently. Transfers are also colored according to the workflow that initiated them. The BLAST workflow has a shared input dataset composed of multiple files, totalling 8GB. This graph demonstrates that Confuga benefits from previous work replicating files by starting the second workflow's jobs immediately on all storage nodes.

The bottom graph also has two sequential runs of the same BLAST workflow but additionally has a BWA workflow (light gray) running concurrently with the first BLAST workflow. The BWA workflow has an input data set of 32GB which is split 274 times. You can see the split done by SN_{11} at 00:15. This graph demonstrates that Confuga is able to run two workflows with disjoint data sets concurrently with data locality and without significantly displacing each other.

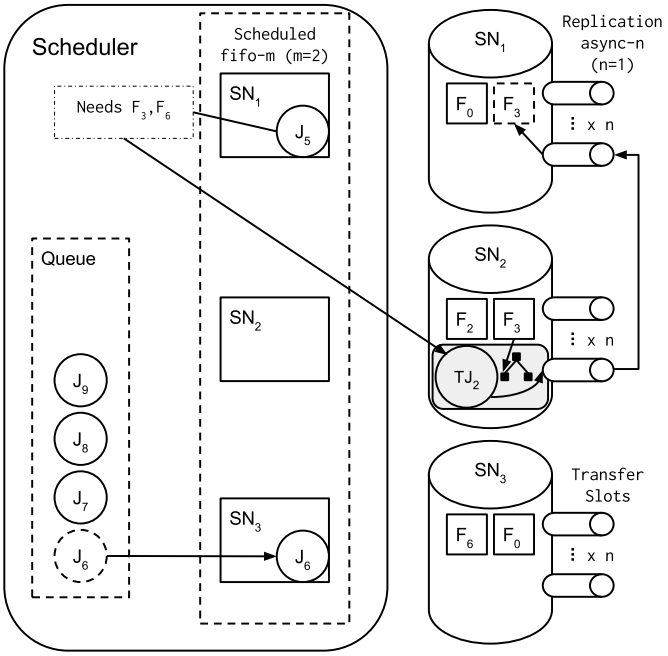

Fig. 8. **Confuga Job Scheduler** This diagram shows the two scheduler parameters we are manipulating in this section: `fifo-m` and `async-n`. This figure shows `m = 2` and `n = 1`. `fifo-m` limits the number of jobs, `m`, which may be in the "scheduled" state, where the job has been assigned a storage node for execution and may begin replicating missing dependencies. So, J_7 may not be scheduled until either J_5 or J_6 leaves the scheduled state and is dispatched. `async-n` limits the number of transfers to and from a storage node. Two missing dependencies of J_5 need to be replicated to SN_1: F_3 and F_6. F_3 is currently being replicated to SN_1, via "Transfer Job" TJ_2 (a Transfer Job is simply a job which transfers a file). F_6 will wait to be replicated until both SN_1 and SN_3 have a free transfer slot ($n = 1$).

and *asynchronous*. The scheduler performs synchronous transfers by serially replicating the dependencies for a job until finished. This is considered a *stop-the-world* scheduling operation as no other scheduling tasks are performed until the transfers are complete. Asynchronous transfers free the scheduler to work on other tasks but require more bookkeeping. Figure 6 visualizes the two strategies.

Synchronous transfers use the blocking `thirdput` RPC [14] for directing a source storage node hosting a replica of the desired file to perform a streaming put of the replica to the target storage node. In our evaluation, we refer to this replication strategy as **sync**.

Confuga implements asynchronous transfers using **transfer jobs** which are regular jobs that execute operations within the storage node file system. This allows for the scheduler to reliably start and track the progress of the asynchronous transfer in a distributed context. Transfer jobs may execute concurrently with other types of jobs. When replicating a file to another storage node, the transfer job performs a `putfile` RPC [14] on the target storage node. While the transfer job is executing, the Confuga scheduler regularly performs a `stat` RPC on the target storage node's developing copy of the replica. This replication strategy is referred to as **async**.

At this time, transfers in Confuga are *directed*; the scheduler controls the movement of all files. This allows Confuga to completely control load on the network. Storage nodes do not independently initiate any transfers. Analyzing the potential benefits of *undirected* transfers via storage nodes independently pulling job dependencies is subject of future work.

B. Constraining Concurrent Job Dispatch

Before a job can be dispatched to a storage node for execution, Confuga must replicate any missing dependencies on the storage node. This results from our chosen semantics `read-on-exec`: each dependency is loaded and mapped into the job's namespace prior to execution. We use the term *scheduled* to refer to the state of jobs which have been assigned to a storage node but the dependencies are not yet all replicated. The scheduler forms decisions about file transfers considering only jobs in the scheduled state.

In early designs of Confuga, we optimistically scheduled jobs on all available storage nodes in one phase of the scheduler. The scheduler would then move on to replicating necessary dependencies for all of these scheduled jobs. For some workflows, this has not been the best default approach as workflows rarely fully utilize the cluster (due to replication and job execution time). Instead, it is useful to conservatively limit the number of *scheduled* jobs that the Confuga scheduler considers at a time. This allows some jobs to execute sooner and enables future jobs to possibly reuse the same nodes that become available. Additionally, each scheduled job uses more of the cluster network for replicating its dependencies.

We refer to this scheduling strategy as **`fifo-m`**, where `m` is the maximum number of jobs in the scheduled state at any time. The early optimistic scheduler corresponded to `fifo-inf`, which practically limits the scheduler to having up to one scheduled job for each storage node (so `fifo-inf` is equal to `fifo-j` where `j` is the number of storage nodes). For `sync` transfers, `fifo-1` is always used because transfers are blocking operations, serially performed by the scheduler. There is no benefit to increasing `m` as the scheduler cannot concurrently replicate dependencies for more than one scheduled job.

C. Constraining Concurrent Transfer Jobs

Once a job is in the scheduled state, the scheduler attempts to replicate any missing dependencies to the storage node it is assigned to. When using asynchronous transfers, a greedy approach would immediately dispatch transfer jobs for all missing dependencies. When there are several large dependencies, this would result in the target storage becoming overloaded or in the cluster network becoming saturated.

Confuga allows for controlling the number of concurrent transfer jobs by assigning `n` *transfer slots* to each storage node. This prevents a node from becoming the target or source of more than `n` concurrent transfer jobs. So the scheduler must wait to replicate any missing dependencies for a scheduled job until the storage node has free transfer slots available. Likewise, a source for a popular replica cannot be overloaded by more than `n` transfers. We refer to this scheduling policy as **async-n**. Figure 8 shows the Confuga scheduler with the `fifo-m` and `async-n` configurations.

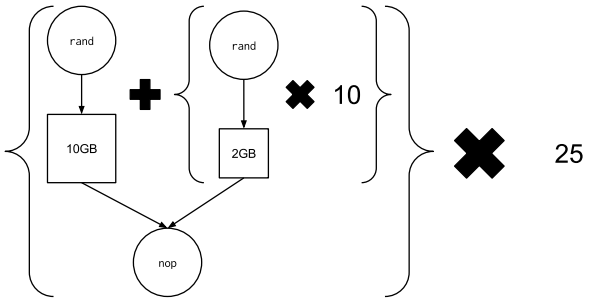

Fig. 9. Transfer heavy workflow used to stress the Confuga scheduler.

D. Evaluation

We have evaluated Confuga's scheduling parameters using the workflow visualized in Figure 9. The goal of this workflow is to stress the Confuga scheduler with short jobs and several large data dependencies. The workflow has 25 instances of a producer and consumer pipeline. Each consumer (`nop`) receives 30GB of data from 11 producers. Figures 10, 12, 11 show the results for running this workflow with several different configurations of the scheduler.

The workflow is limited by the ability to transfer files between nodes, so an increase in the cluster bandwidth leads to a decrease in execution time. Increasing `fifo-m` has the most significant impact on the cluster bandwidth, across all configurations of `async-n`, except `async-inf`. An unlimited number of concurrent transfers on a storage node appears to hit a cluster bandwidth limit around 150 MB/s, even for increasing `fifo-m`. On the other hand, `async-1` achieves the best performance for all configurations of `fifo-m`, allowing a single transfer to saturate the link for a storage node.

For the bandwidth of individual transfers, increasing the concurrency of transfers on a storage node (`async-n`) or on the cluster (`fifo-m`) has a negative impact. Despite this, the utilization of the cluster network increases. This allows for the system as a whole to accomplish more, even though individual transfers are slower.

We conclude from these experiments that a directed and controlled approach to managing transfers on the cluster is essential for achieving performance. For example, enforcing a limit on transfers to one per storage node offered a 228% increase in average cluster bandwidth and a 23% reduction in workflow execution time (`fifo-inf/async-1` vs. `fifo-inf/async-inf`). This method of controlling load on the cluster, allowing for efficient disk and network utilization, is made possible by Confuga exploiting the explicit job namespace descriptions from the workflow and by Confuga limiting the consistency semantics for jobs.

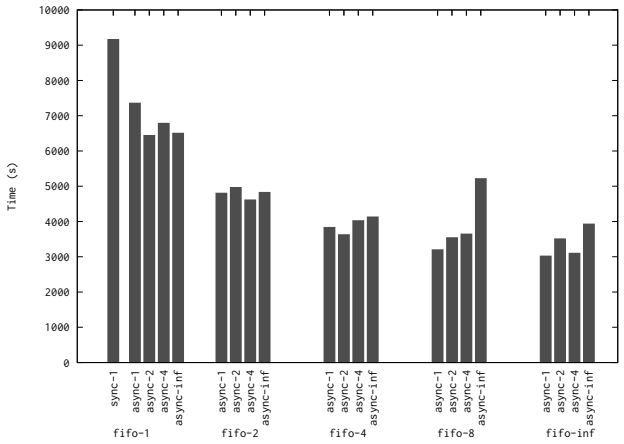

Fig. 10. Time to complete.

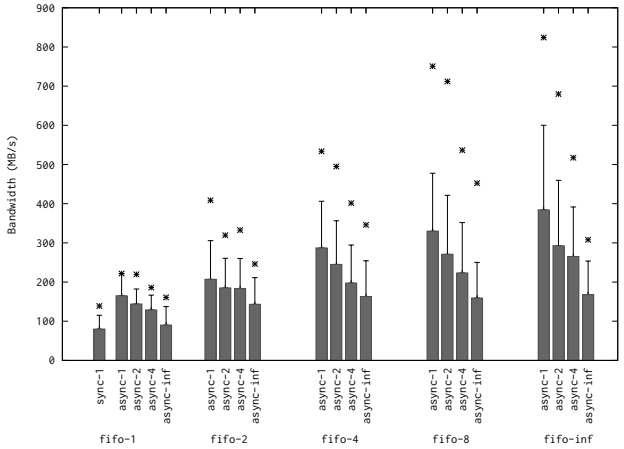

Fig. 11. Cluster Transfer Bandwidth. Bars, whiskers, and stars are respectively average, standard deviation, and maximum.

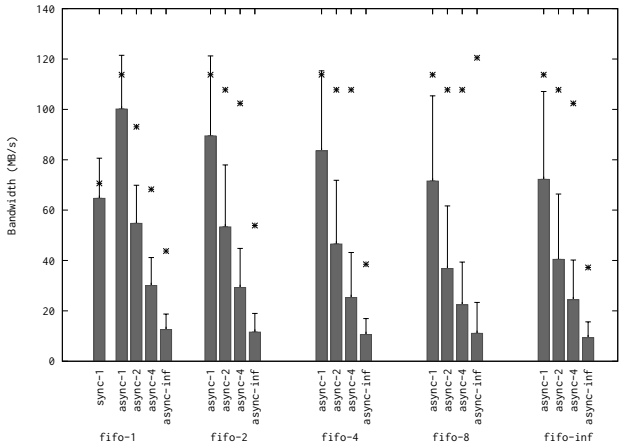

Fig. 12. Individual Transfer Bandwidth. Bars, whiskers, and stars are respectively average, standard deviation, and maximum.

VII. Related Work

Distributed file systems like NFS [15] and AFS [16] are often used in multi-user systems with reasonably strict POSIX compliance. AFS is notable for introducing write–on-close, so other clients will not see changes to a file until file close. Other POSIX extensions for high-performance computing have been proposed [17] allowing batch metadata operations and explicitly relaxing certain consistency semantics.

Still, all parallel cluster file systems suffer from the metadata bottleneck [18], [19], [20]. 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 is a common feature in file systems, including Venti [21] and HydraFS [22]. Confuga uses both CAS and GUIDs to empower storage nodes to self-allocate a filename in the replica namespace without head node intervention. File deduplication is a secondary benefit.

Active Storage [5] 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 manufacturer interest, this idea has not been implemented. Other work on Active Storage has centered around the use of workflow abstractions with workflow managers tightly coupled to the storage system. Map-Reduce [1] is an abstraction used in Hadoop [23] 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.

Object based storage devices (OSD) [24] 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 [25]. OSD has become an integral component of numerous file systems including Lustre [26] and Panasas [27]. Computation on Lustre storage servers has also been supported [28] to allow client programs to have direct access to data, and in [29] as a user-space solution.

VIII. Acknowledgments and Availability

This work was supported in part by National Science Foundation grant OCI-1148330. Confuga's source code is distributed under the GNU General Public License. See the URLs below for source code and workflows used in this paper.

`http://ccl.cse.nd.edu/`
`https://github.com/cooperative-computing-lab/cctools/tree/papers/confuga-ccgrid2015`

References

[1] Dean, Jeffrey and Ghemawat, Sanjay, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, Jan. 2008.

[2] Zaharia, Matei and Chowdhury, Mosharaf and Franklin, Michael J and Shenker, Scott and Stoica, Ion, "Spark: Cluster Computing With Working Sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[3] Malewicz, Grzegorz and Austern, Matthew H. and Bik, Aart J.C and Dehnert, James C. and Horn, Ilan and Leiser, Naty and Czajkowski, Grzegorz, "Pregel: A System for Large-scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10, 2010.

[4] Zaharia, Matei and Borthakur, Dhruba and Sen Sarma, Joydeep and Elmeleegy, Khaled and Shenker, Scott and Stoica, Ion, "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10, 2010.

[5] 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*, 1998.

[6] M. Szeredi, "FUSE: Filesystem in Userspace. 2005." [Online]. Available: http://fuse.sourceforge.net

[7] Crockford, Douglas, "The application/json Media Type for JavaScript Object Notation (JSON)," 2006.

[8] 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*, 2009.

[9] Albrecht, Michael and Donnelly, Patrick and Bui, Peter and Thain, Douglas, "Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids," in *1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, 2012.

[10] D. Thain, C. Moretti, and J. Hemmes, "Chirp: a practical global filesystem for cluster and grid computing," *Journal of Grid Computing*, 2009.

[11] D. Thain and M. Livny, "Parrot: An application environment for data-intensive computing," *Scalable Computing: Practice and Experience*, 2005.

[12] Howard, John H. and Kazar, Michael L. and Menees, Sherri G. and Nichols, David A. and Satyanarayanan, M. and Sidebotham, Robert N. and West, Michael J., "Scale and Performance in a Distributed File System," *ACM Trans. Comput. Syst.*, Feb. 1988.

[13] Isard, Michael and Prabhakaran, Vijayan and Currey, Jon and Wieder, Udi and Talwar, Kunal and Goldberg, Andrew, "Quincy: Fair Scheduling for Distributed Computing Clusters," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09, 2009.

[14] Thain, Douglas and Moretti, Christopher, "Efficient Access to Many Small Files in a Filesystem for Grid Computing," in *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, 2007.

[15] 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*, 1985.

[16] 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)*, 1988.

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

[18] 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*, 2011.

[19] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-file access in parallel file systems," in *IEEE International Symposium on Parallel & Distributed Processing*, 2009.

[20] 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*, 2000.

[21] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage." in *FAST*, 2002.

[22] 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*, 2010.

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

[24] 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*, 1997.

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

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

[27] 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*, 2004.

[28] 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.

[29] 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*, 2007.