Explicit Control in a Batch-Aware Distributed File System

Abstract

We present the design, implementation, and evaluation of the Batch-Aware Distributed File System (BAD-FS), a system designed to orchestrate large, I/O-intensive batch workloads on remote computing clusters distributed across the wide area. BAD-FS consists of two novel components: a storage layer which exposes control of traditionally fixed policies such as caching, consistency, and replication; and a scheduler that exploits this control as needed for different users and workloads. By extracting these controls from the storage layer and placing them in an external scheduler, BAD-FS manages both storage and computation in a coordinated way while gracefully dealing with cache consistency, fault-tolerance, and space management issues in an application-specific manner. Using both microbenchmarks and real applications, we demonstrate the performance benefits of explicit control, delivering excellent end-to-end performance across the wide-area.

1 Introduction

Traditional distributed file systems, such as NFS [45] and AFS [26], are built on the solid foundation of empirical measurement. By studying expected workload patterns [8, 36, 41, 46, 52], researchers and developers have long been able to make appropriate trade-offs in system design, thereby building systems that work well for the workloads of interest.

Most previous distributed file systems have been targeted at a particular computing environment, namely a collection of interactively-used client machines. However, as past work has demonstrated, different workloads lead to different designs (*e.g.*, FileNet [16] and the Google File System [23]); if assumptions about usage patterns, sharing characteristics, or other aspects of the workload changes, one must reexamine the design decisions embedded within distributed file systems.

One area of increasing interest is that of "batch" workloads. While batch workloads have long been popular among scientists, they now are common across a broad range of important and often commercially viable application domains, including genomics [4], video production [47], simulation [10], document processing [16], data mining [2], electronic design automation [15], financial services [38], and graphics rendering [31].

Batch workloads minimally present the system with the set of jobs that need to be run and perhaps some ordering among them; in many environments, the approximate run times and information about I/O requirements are also known in advance of run-time. Such knowledge is encapsulated within a workflow manager (also called a scheduler), which schedules jobs across the nodes of the system so as to maximize throughput, and handles failures in job execution through retry or other techniques [22].

Batch workloads are typically run in controlled local-area cluster environments [33, 53]. However, organizations that have large workload demands increasingly need ways to share resources across the wide area, to lower costs and increase productivity. A simple approach to accessing resources across the wide-area is to run a local-area batch system across multiple clusters that are spread out across the wide-area, and to use a distributed file system as a backplane for data access.

Unfortunately, this approach is fraught with difficulty, largely due to the way in which I/O is handled. The primary problem with using a traditional distributed file system for I/O in this domain is in its approach to *control*: many decisions are made *implicitly* within the file system with regards to the caching, consistency, and fault tolerance of data. While these decisions are reasonable for the workloads that these file systems were designed for, they are ill-suited for a wide-area batch computing system. For example, to minimize bandwidth utilization across the wide-area, the system must carefully utilize the cache resources (*e.g.*, the disks) of remote clusters; however, such caching decisions are buried deep within distributed file systems, thus preventing such control.

To mitigate these problems, and thus enable the utilization of remote clusters for I/O-intensive batch workloads, we introduce the Batch-Aware Distributed File System (BAD-FS). BAD-FS differs from traditional distributed file systems in its approach to control: BAD-FS exposes decisions commonly hidden inside of a distributed file system to an external workload-savvy scheduler. BAD-FS leaves all consistency, caching, and replication decisions to an external scheduler, thus enabling *explicit* and application-specific control of file system behavior.

The main reason to migrate control from the file system to the scheduler is *information* – the scheduler has intimate knowledge of the workload that is running and can exploit said knowledge to improve performance and streamline failure handling. The combination of workload information and explicit control of the file system leads to three distinct benefits over traditional approaches.

First, explicit control of the file system by the batch scheduler improves performance. By carefully managing remote cluster disk caches in a cooperative fashion, and by scoping I/O such that only needed data is transported across the wide-area, BAD-FS minimizes wide-area file system traffic and hence improves job throughput.

Second, explicit control combined with workload knowledge improves failure handling. The scheduler can determine whether to make replicas of output data based on the cost of generating the data, and not in a blind fashion as is typical in file systems. Data loss is treated uniformly as a performance problem – the scheduler has the ability to regenerate lost files by rerunning the application that generated it – and hence replication is employed only when the cost of regeneration is high.

Third, explicit control simplifies implementation. For example, because the scheduler has exact knowledge of data dependencies between jobs, BAD-FS provides a cooperative cache but does not implement a cache consistency protocol; the scheduler ensures proper access ordering among processes. Previous work in distributed file systems has demonstrated the difficulties of building a more general cooperative caching scheme [6, 11].

BAD-FS achieves these ends while maintaining site autonomy and support for unmodified legacy applications. Both of these practical constraints are important for acceptance in wide-area batch computing environments.

We demonstrate the benefits of explicit control via experimentation on our prototype implementation of BAD-FS. Through controlled microbenchmarks, we demonstrate that BAD-FS can reduce wide-area I/O traffic by an order of magnitude, and can proactively replicate data to obtain high performance in spite of remote failure. With real applications, we demonstrate the practical benefits of our system: I/O-intensive batch jobs can be run upon remote resources both easily and with high performance.

The rest of this paper is organized as follows. In Section 2, we describe our assumptions about the expected environment and workload, and in Section 3, we discuss the architecture of our system. In Section 4, we present our experimental evaluation. In Section 5, we examine related work, and finally in Section 6, we conclude.

2 Background

In this section, we describe the setting for BAD-FS. We present the expected batch workloads and the computing environment available to users with such needs. We then discuss the problem of executing such workloads with conventional tools from the perspective of an end user.

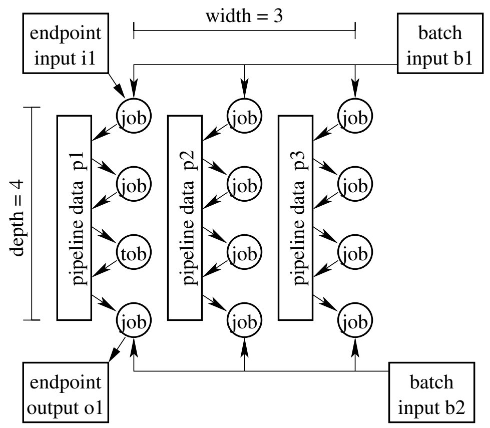

Figure 1: **Workload Structure.** *A typical batch-pipelined workload is depicted. A single pipeline represents the logical work that a user wishes to complete, and is comprised of a series of jobs. Users often assemble many such pipelines into a batch to explore variations on input parameters or other input data.*

2.1 Workloads

We now define the expected data-intensive, high-throughput workload in more detail, basing assumptions on our recent work in batch workload characterization [7]. As illustrated in Figure 1, these workloads are composed of multiple independent pipelines. Each pipeline contains a group of sequential processes that communicate with the preceding and succeeding processes via private data files. A workload is generally submitted in large batches with all of the pipelines incidentally synchronized at the beginning, but each pipeline is logically distinct and may correctly execute at a different rate than its siblings. We call these batches of pipelines a *batch-pipelined* workload. [1]

One of the key differences between a single application and a batch-pipelined workload is file sharing behavior. For example, when many instances of the same application are run, the same executable and potentially many of the same input files are used. We characterize the sharing that occurs in these batch-pipelined workloads by breaking I/O activity into three categories: *endpoint*, which represents the non-shared input and final output, *pipeline-shared*, which is shared in a write-then-read fashion within a single pipeline, and *batch-shared*, which is comprised of input I/O that is shared across multiple pipelines.

When considered in isolation, a single pipeline does not present an excessive load for a computing system. However, when considered in aggregate, such workloads produce I/O traffic that is handled poorly by traditional systems. We will explore this issue fully below, but a brief example is suitable here.

[1] The term “pipeline” is used generically. Each job in a pipeline communicates via files. The jobs are *not* connected by Unix-style pipes.

One of the applications we studied is known as BLAST, a commonly used genomic search program. BLAST consists of only a single stage pipeline, but each process streams through a shared large database (*e.g.*, 1 GB) looking for string matches. Consider running the BLAST workload on a computing cluster of 100 nodes equipped with a conventional distributed file system such as AFS or NFS. With cold caches, all 100 nodes will individually (and likely simultaneously) hit the file server with this large request, resulting in a long blocking read as 100 GB of data are transferred over the wide area network. Once the caches are loaded, each node will run at local disk speeds, if the database can fit in cache. Each node assumes a process may perform arbitrary read/write operations, so it must keep contact with the home file server to ensure file system consistency; if this connection is lost, the node must pause or abort to avoid data inconsistency.

In contrast, a distributed system such as BAD-FS has a global view of the hardware configuration and workflow structure; it can execute such workloads much more efficiently by copying the database a single time over the wide area and sharing or duplicating the data at the remote cluster. Further, explicit knowledge of sharing characteristics permits such a system to dispense with the expense and complexity of consistency checks while allowing nodes to continue executing, even if disconnected from the home server.

2.2 Environment

Although wide-area sharing of untrusted and arbitrary personal computers is a possible platform for batch workloads [48], we believe that a better platform for these types of throughput-intensive workloads is one or more clusters of managed machines, spread across the wide area. We assume that each machine within a cluster has processing, memory, and local disk space available for remote users, and that each cluster exports their resources through some type of CPU sharing system. The obvious bottleneck of such a system is the wide-area connection, which must be managed carefully to ensure high performance. Note that for simplicity, we focus our efforts within this paper on the case of a single remote cluster; we plan to address the issues that arise in multiple cluster settings in future work.

We sometimes refer to this more organized, less hostile, and well managed collection of clusters as *c2c (cluster-to-cluster)* computing, in contrast to widely popular peer-to-peer (p2p) systems. Although the p2p environment is appropriate for many uses, there is likely to be a more organized effort to share computing resources within corporations or other organizations. Basic assumptions about machine behavior, including stability, performance, and trust, are different. That said, we believe that much of the p2p systems technology that develops is likely to be directly applicable to the c2c domain.

We also make the practical and important assumption that each site that participates has local autonomy over its resources. Autonomy has two primary implications on the design of BAD-FS. First, although an application may be able to use remote resources at a given time, these resources may be taken away at a moment's notice, perhaps to be given to a "more important" local user. Thus, a system that is built to exploit remote resources must be able to tolerate unexpected resource "failure", *i.e.*, actual hardware or software failure, or even a prioritized preemption by the owning site. Second, autonomy prohibits the deployment of arbitrary software within the remote cluster; in designing BAD-FS, we must assume only the bare minimum of software is available (namely, a remote batch system which enables BAD-FS to run jobs on the cluster). Mandating that a single distributed file system be used for all participating remote clusters is not a viable solution.

Finally, we assume that the jobs run on these systems cannot be modified. There are two reasons to make this assumption. First, in our experience, many scientific applications are the product of years of fine-tuning, and when complete, are viewed as "untouchable". Second, ease of use is important; the less work the user has to do to run their jobs, the better.

2.3 Example Usage

We now consider a user who wishes to run a batch-pipelined workload in this environment. After the user has developed and debugged the application on their home system, they are ready to run hundreds or thousands of instances of their application on all available computing resources, using a remote batch execution system such as Condor [33], LSF [53], PBS [50], or Grid Engine [49].

Each instance of their application is expected to use much of the same input data, while varying parameters and other small inputs. The necessary input data begins on the user's *home storage server* (*e.g.*, an FTP server), and the output data, when generated, should eventually be committed to this home server.

The state of the art solution presents a user with two options for running a workload. The first option is to simply submit the workload to the remote batch system. With this option, all input and output occur on demand back to the home storage device as the jobs run. While this approach is simple for the user, the performance of a data-intensive application will not be acceptable for two reasons. First and most importantly, wide-area network bandwidth and latency limitations are not sufficient to handle simultaneous requests from many data-intensive applications running in parallel. Second, all application I/O is directed back to the home site, including temporary data that is not needed after the computation is complete.

The second option is for the user to manually configure their system to replicate their data sets in the remote

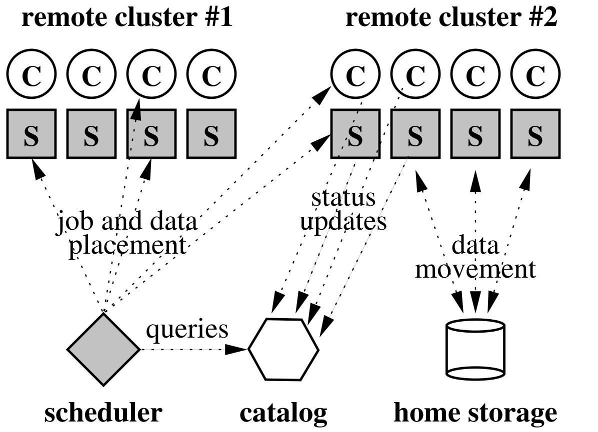

Figure 2: **System Architecture.** *Circles are compute servers, which execute batch jobs. Squares are storage servers, which hold cached inputs and temporary outputs. Both types of servers report to a catalog server, which records the state of the system. The scheduler directs the entire system by configuring storage devices and submitting batch jobs. The grayed shapes are novel elements in our system design, while the white shapes are standard components found in a batch system.*

environment. This requires the user to identify the necessary input data, transfer the data to the remote site using a tool such as FTP, log into the remote system, unpack the data in an appropriate location, configure the application to recognize the correct directories, submit the jobs, and deal with any failures that occur. The entire process must be repeated whenever more data needs to be processed, new batch systems become available, or existing systems no longer have capacity to offer to the user. As is obvious from the description, the process is labor-intensive and error-prone (and yet today, many users of such systems go to these lengths simply to run their jobs).

BAD-FS solves these problems by creating a personal data-intensive computing environment on the basic substrate of one or more batch systems. BAD-FS is responsible for dynamically deploying a set of services that identify the combined compute and storage resources in a cluster and export them in manner that manages resources efficiently and hides faulty behavior, all while remaining easy to use.

3 System Architecture

In this section, we present the architecture and implementation of BAD-FS. Recall that the main goal of the design of BAD-FS is to export sufficient control to a remote scheduler to deliver improved performance and better fault-handling for I/O-intensive batch workloads run on remote clusters. Figure 2 summarizes the architecture of BAD-FS, with the novel elements shaded gray.

BAD-FS is structured as follows. Two types of server processes manage local resources. A *compute server* exports the ability to transfer and execute an ordinary user program on a remote CPU. A *storage server* exports access to disk and memory resources via remote procedure calls that resemble standard file system operations such as open, close, read, and write. It also permits remote users to allocate space for various purposes via an abstraction called a *container*. Both types of servers periodically report themselves to a *catalog server*, which summarizes the current state of the system. A *scheduler* periodically examines the state of the catalog, considers the work to be done, and assigns jobs to compute servers and data to storage servers. The scheduler may obtain data, executables, and inputs from any number of external storage sites. For simplicity of exposition, we assume the user has all the necessary data stored at a single *home storage server* such as a standard FTP server.

From the perspective of the scheduler, compute and storage servers are logically independent. A specialized device might run only one type of server process: for example, a diskless workstation runs only a compute server, while a storage appliance runs only a storage server. However, a typical workstation or cluster node has both computing and disk resources and thus runs one instance of each server. The BAD-FS scheduler considers these two servers to be independent resources, only noting that they have the same network address and thus are "close" to each other.

As we noted above, BAD-FS may be run in an environment with multiple owners and a high rate of failure. To reflect this environment, we assume the following pessimistic properties about the components of the system. Both storage and compute servers may be owned by various individuals or institutions, not necessarily the same as the owner of the scheduler(s). Thus, such servers are free to evict jobs or data with no warning. Because the system state may change rapidly, the summary and membership information present in the catalog may be stale. The scheduler must be prepared to discover that servers it attempts to harness may not be in the state that it expects. Finally, we expect to encounter network disconnections and crashes, perhaps followed by reboots, of any server. The scheduler must operate in such a way that all resource allocations are recorded persistently so that appropriate cleanup may be performed when possible. For this, we rely extensively on transaction protocols and logging, as described below.

BAD-FS makes use of several standard components found in most batch systems. Namely, our compute servers are Condor *startd* processes, while our catalog is the Condor *matchmaker* [33, 39]. All servers advertise themselves to the catalog via the ClassAd resource description language. This semi-structured language permits servers to describe arbitrary properties about themselves with name-value pairs and simple data structures such as lists and sets. Servers typically advertise their name, address, current users, available capacity, and so forth. The scheduler (or the user) may query the catalog for a complete dump of the system, or may perform complex queries based on properties of the respondents.

We now discuss the main components of BAD-FS. First, we describe the storage servers and related software, and then describe the BAD-FS scheduler. Finally, we discuss other practical issues.

3.1 Storage Servers

The storage layer has the core responsibility of exporting storage resources of the remote sites in a manner that allows efficient remote management by the scheduler. The storage layer is composed of a set of *storage servers*, each of which can be configured to perform various I/O tasks through an abstraction called a *container*; the container is the key abstraction through which the scheduler controls remote storage resources. To simplify the use of the storage layer by unmodified applications, BAD-FS also includes an *interposition agent* (described in Section 3.1.2) which converts the I/O operations of user's jobs into operations on the storage layer at runtime.

3.1.1 Containers

A storage server does not have a fixed policy for managing its storage. Rather, it makes several policies available to external users (or more likely, to the BAD-FS scheduler), who may carve up the available space for caching, buffering, or other tasks as they see fit. Control is exposed through an allocation unit called a *container*. A container is an allocation of storage space with a name, a lifetime, and a policy that controls how the space is internally managed. The lifetime is a lease, which upon expiration, causes the entire container to be deleted atomically.

When a container is created, the caller may select what algorithm is used to manage its contents. Currently, the BAD-FS storage server implements three distinct container policies: the scratch, standalone cache, and cooperative cache containers.

A *scratch container* is a self-contained read-write file system for storing temporary data such as a pipeline file between two processes. Typically, a scratch container is used as a waypoint for data in transit. For example, a job may write output data into a scratch container, where it sits passively until retrieved by a scheduler. When so directed by a scheduler, a storage server can duplicate a scratch container elsewhere by contacting a peer server to allocate space and transfer data.

A *standalone cache container* is a read-only view of another storage server. When created by the scheduler, a container is given the name of the target server and path, a caching policy (*i.e.*, LRU or MRU), and a maximum storage size.

A *cooperative cache container* is a read-only view of another storage server that also works in concert with peer storage servers. When created by the scheduler, it takes the same arguments as a standalone cache container, but also accepts the name of a catalog server to consult for the names of its peer servers. There exist a number of algorithms [14, 37] for managing a cooperative cache, but it is not our intent here to explore the range of these algorithms. Rather, we will describe a suitable algorithm for this system and explain how the scheduler can manage workloads in concert with the cooperative cache.

The cooperative cache is essentially a distributed hash table [24, 32]. The keys in the table are filenames and block numbers, and the values are the corresponding data blocks. Each peer in the cache has the same hash function. To divide the key space, each peer periodically probes the catalog server of the list of its peers. Each then sorts the list by network address and assigns ranges of the key space as leaves in a binary tree. If the number of peers is a power of two, each will store the same amount of cache space. If not, some will have up to twice as much as the others. Although this scheme is asymmetric with respect to space, it minimizes the reassignment of cache space as peers enter and leave the cache. Data within each participating node of the cooperative cache is managed in a global LRU-like fashion.

This approach to cooperative caching has two important differences from previous work [14, 17]. First, because application data dependencies are completely specified by the scheduler, we do not need to implement a cache consistency scheme. This design decision greatly simplifies our implementation; previous work has demonstrated the many difficulties of building a more general cooperative caching scheme [11]. Second, unlike previous cooperating caching schemes that manage cluster memory in a global fashion, our cooperative cache stores data in the local *disks* of each remote node. Although managing memory caches cooperatively could also be advantageous, the most important performance optimization to make in our environment is to avoid data movement across the wide-area link; managing remote disk caches effectively is the simplest and most effective way to meet this goal.

Note that containers export only a certain level of control to the external scheduler. Namely, by creating and deleting containers, the scheduler can control which data sets reside in the disk space of the remote cluster. However, the caches retain control over per-block decisions, *i.e.*, if more blocks are accessed than fit in a container, the cache will make a local replacement decision. Of course, if the scheduler is careful in space allocation, such replacements will occur rarely. In general, we have found this separation of coarse-grained and fine-grained decision making to be suitable for our expected workloads.

3.1.2 Interposition Agents

In order to permit unmodified applications to make use of BAD-FS storage servers, an *interposition agent* [27] transforms standard POSIX I/O operations into requests to storage servers. The agent is programmed by the scheduler with a mapping (a *mount list*) from logical file names to physical containers.

Together, the interposition agent and the container abstraction provide a high degree of failure detection that can be hidden from the application and the end user. If a container no longer exists, either due to accidental failure or deliberate preemption, a storage server returns a unique "container lost" error to the agent. This error is deliberately distinct from "file not found," which indicates that a named file does not exist in a valid container; such a value could occur in a valid program searching for a file in multiple places, or it could indicate that the user misconfigured the input data. Thus, a "file not found" is passed directly to the application for consideration. However, a "container lost" is of interest to BAD-FS itself. Upon discovering this error, the agent forcibly terminates the application, indicating that the job could not run correctly in the given environment. This gives the scheduler early and clear indication of failures, and allows it to take recovery actions transparent to the application and user.

Our current interpositioning technique relies on the debugging interface, although many other techniques are suitable [3]. At the execution site, the interpositioning tool runs the target application as a child, halting it at the entry to and exit from all system calls. While the child process is stopped, the interposition agent emulates the desired system calls by copying data in and out of the child. All of the application's I/O state, including the table of open files, current seek pointers, and so forth, is kept in the interpositioning tool. Unlike other tools such as UFO [3], this agent does not perform whole-file fetch at first open. Instead, the agent only services the minimal read or write necessary to satisfy the application. This permits applications to perform random, partial file access to large datasets without transferring unneeded data.

3.2 The Scheduler

The BAD-FS scheduler is designed to take advantage of the explicit control provided by the BAD-FS servers, and thus orchestrates the execution of a batch workload across remote clusters. The scheduler combines dynamic knowledge of the system with static knowledge of the user workload to improve performance and recover cleanly from failures. Specifically, the scheduler reduces traffic across the wide-area by differentiating I/O types and treating them accordingly, carefully manages the contents of remote server caches to avoid thrashing, and replicates remote output data proactively if said data is expensive to regenerate.

```
job a a.condor
job b b.condor
job c c.condor
job d d.condor
parent a child b
parent c child d
volume b1
   ftp://home/data 1 GB
volume p1 scratch 50 MB
volume p2 scratch 50 MB
mount b1 a /mydata
mount b1 c /mydata
mount p1 a /tmp
mount p1 b /tmp
mount p2 c /tmp
mount p2 d /tmp
extract p1 x
   ftp://home/out.1
extract p2 x
   ftp://home/out.2
```

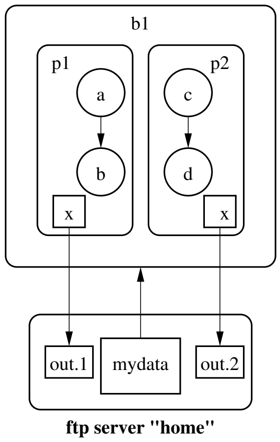

Figure 3: **Workflow Language and Schematic.** *An example workflow is depicted. A directed graph of jobs is constructed via the* `job` *and* `parent` *keywords, and the file system namespace presented to jobs is configured via* `volume` *and* `mount` *directives. The* `extract` *keyword indicates which files must be committed to stable storage at the home storage server upon job completion.*

3.2.1 Workflow Description

We now present how users describe their workloads to the system. Figure 3 shows an example and a schematic rendering of our workflow language. The keyword `job` declares an abstract job name and binds it to a job description file suitable for the virtual batch system. In this example, job `a` is bound to the job description file `a.condor`, which names the executable, input and output files, architecture constraints, and environment variables. The `parent` keyword indicates an ordering between two jobs. In this example, `a` must execute before `b` and `c` before `d`.

More interesting is the manner in which the local namespace of a job is constructed. The `volume` and `mount` directives establish the binding between data sources to the private name space in each job. For example, the declaration of volume `v1` is used to establish the binding from `/mydata` to the nearby storage server; the `mount` commands specify that jobs `a` and `b` share the same `/tmp` directory.

Finally, the workflow provides a way for jobs to differentiate between scratch data space (which is used either privately by a single process or as a method of communication between jobs in a pipeline) and output that must be committed reliably to the home storage server. The `extract` command specifies which files in which volumes should be written home when the job is complete. Optionally, the size of a volume may be given to aid the scheduler in allocating storage space.

We note that users running a large number of interdependent jobs must always express these types of dependencies. To successfully execute thousands of jobs,

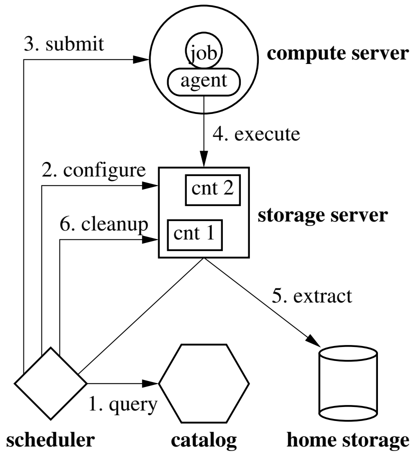

Figure 4: **Running a Job with BAD-FS.** *1. The scheduler queries the catalog for the current system state and decides where to place a job and its data. 2. The scheduler configures a storage server for the job. 3. The job is submitted to the compute server. 4. The job executes, accessing its data via the agent. 5. After the job completes, the scheduler extracts selected output files. 6. The storage server is cleaned up.*

one must create an organized directory structure and determine which jobs use data created by others. Many users currently specify these dependencies by writing shell scripts and makefiles that explicitly control execution order. A workflow language has the advantage that it effectively abstracts what operations need to be done from how those operations are performed, much in the way that relational queries separate what the user wants from how it gets computed [12]. This abstraction allows users to be blissfully unaware of low-level system details (*e.g.*, how failures are handled), while giving the system powerful information about jobs and data.

3.2.2 Basic Operation

The scheduler operates as follows. First, the manager scans the workflow for ready but unassigned jobs and volumes and assigns them to resources. When there is no work left to be assigned, the manager waits to be notified of changes in job state by the batch queue. As jobs complete, children may be dispatched and any unneeded resources cleaned up. Periodically, the scheduler refreshes its model of the system by querying the catalog for a list of resources. An illustration of how a single job is run is depicted in Figure 4.

3.2.3 I/O Scoping

Unlike data in most file systems, all of which is assumed to be equally important, the BAD-FS scheduler has a good understanding of how each file is used and where it is needed. The scheduler can take advantage of this information to reduce data movement across the wide-area. Specifically, the scheduler is aware of which input files are shared ("batch" data), which output files are ephemeral ("pipe" data), and which output files are needed by the user after job completion ("endpoint" data).

Specifically, the scheduler directs all read-write traffic between jobs in a pipeline to a scratch volume, thus keeping said traffic within the remote cluster. The only final output data that must be moved to the home storage server is specified in the `extract` statements in the workflow. We refer to this process as *I/O scoping*, as data only moves to the scope where it is needed. Scoping could instead be approximated manually by rewriting jobs to use `/tmp` for pipeline data; however, this approach increases the burden on users by requiring application change and does not mesh cleanly with our failure handling machinery.

3.2.4 Cache Consistency

With the information expressed in the workflow, the scheduler neatly addresses the issue of cache consistency management. All of the required dependencies between jobs are specified directly in the workflow; by running jobs so as to meet these constraints, we avoid the need to implement a cache consistency protocol among the BAD-FS servers. However, this does leave the burden on users to ensure their workflows do not contain data sharing that is not described to the scheduler; in the future, we plan to add a mechanism to the servers to detect this condition and warn users of any unexpected behavior.

3.2.5 Capacity-Aware Scheduling

As the scheduler is also aware of how much cache space is available at the remote site, it can ensure that the jobs that are running utilize that space effectively. Specifically, the scheduler takes pains to avoid overflowing remote cache resources and thrashing the remote cache, an approach we call *capacity-aware scheduling*.

The algorithm works as follows. The scheduler retrieves the current state of the system from the catalog, and examines the available storage. The scheduler then examines all of the unexecuted jobs whose parents have completed, and selects the one with the least unfulfilled storage needs, whether pipe or batch. If the scheduler is able to satisfy all of the volumes needed by the job simultaneously, then the scheduler allocates the necessary containers from one or more storage servers, configures them for the given job, and submits the job to the batch system. If there is still free storage remaining, the scheduler selects the next job with the smallest unfulfilled requirements, and continues on. If there are no jobs to execute or not enough available space, then the scheduler waits for a job to complete, more resources to be added to the system, or a failure to occur, and reacts appropriately.

The scheduler does not attempt any high degree of optimization in the ordering of resources allocated, although such a scheduler could be built. Its primary job is to avoid disasters of resource allocation, such as overcommitment of a cache by jobs competing for the same physical resources. As Lampson said, “There probably isn’t a ’best’ way to build the system ... ; much more important is to avoid choosing a terrible way” [30]. By selecting the job with the smallest storage requirements, the scheduler selects an allocation sequence that avoids deadlock, if one exists. If no allocation can be made to move the workflow forward, the scheduler may be configured to either wait for further resources to be added to the system, or to stop the workflow. Likewise, the scheduler is capable of aborting a workflow in progress, hence removing jobs and freeing storage.

3.2.6 Handling Failures

Finally, a key component of the scheduler is found in how it makes BAD-FS robust to failures. The scheduler can handle failures of batch jobs, storage servers, the catalog, and the scheduler itself. To accomplish this, it keeps a log in persistent storage and uses a transactional interface to the job queue and storage containers. If the scheduler fails, it recovers from the log and resumes operation without losing jobs or storage containers.

BAD-FS handles failures of computation and storage by waiting for passive indications, and then conducting active probes as necessary. For example, if a job returns to the scheduler with an abnormal exit code indicating an I/O failure (generated by the interposition agent), it suspects that the storage servers housing one or more of the containers assigned to the job are faulty. The scheduler then probes the servers to check for the containers. If all containers are healthy, then it is assumed the job encountered transient communication problems and is simply resubmitted. However, if the containers have failed or are unreachable for some period of time, the containers are assumed lost.

When a container is lost, the scheduler deletes it and checks all processes that have or will interact with volumes in that container. Clearly, currently running processes that rely on that volume for input data must be stopped; these processes will be restarted later when their input data is restored. However, the processes that *wrote* to this volume may also need to be restarted; that is, BAD-FS needs to restart the jobs that created the lost files needed by the stopped jobs.

In order to avoid these expensive restarts of a workflow, the scheduler may direct replication of scratch volumes as stages of a pipeline complete. Given that the importance of replicating a volume depends upon both the probability of failure and the execution time of the jobs creating this data, the scheduler performs a cost-benefit analysis at run-time to determine when a volume should be replicated. The result of our cost-benefit approach is a replication strategy that is tuned to the needs of the workload and expected failure characteristics of the run-time environment, instead of a naive scheme that replicates all data in a uniform manner.

Our initial cost-benefit algorithm works as follows. To determine the cost of replicating a volume, the scheduler tracks the average time necessary to replicate a scratch volume from one server to another. This cost is initially assumed to be zero in order to trigger the scheduler to perform at least one replication and subsequent measurement. To determine the benefit of replicating a volume, the scheduler tracks the number of job and storage failures and computes the mean-time-to-failure across all devices in the system; this value is initially assumed to be one day, as suggested by Long*et al.* [34]. The benefit of replicating a volume is then the sum of the run times of those jobs completed so far in the applicable pipeline multiplied by the probability of failure. If the benefit exceeds the cost, then the scheduler replicates the container on another storage server as insurance against failure. If the original container fails, the scheduler restarts the pipeline using the saved copy.

3.3 Practical Issues

One of the primary obstacles to deploying a new distributed system is the need for a friendly administrator. Whether deploying an operating system, a file system, or a batch system, the vast majority of such software requires a privileged user to install and oversee the software. Such requirements make many forms of distributed computing a practical impossibility; the larger and more powerful the facility, the more difficult it is for an ordinary user to obtain administrative privileges. One of the strengths of BAD-FS is its ability to export *existing* resources to the users in a more coordinated and palatable form than the underlying resource. To this end, BAD-FS is packaged as a *virtual batch system* that can be deployed over an existing batch system. This technique is patterned after the “glide-in job” described by Frey *et al.* [21], and is similar in spirit to research in recursive virtual machines [19] and overlay networks [5].

Suppose an ordinary user has access to an existing batch system. BAD-FS bootstraps itself on these systems, relying only on the ability to queue and run a self-extracting executable package. BAD-FS includes the storage server, a compute server, and the interposition agent inside of this package. Once deployed to an execution site, the package is expanded, and the servers are started and make themselves known to the catalog server. The BAD-FS scheduler can then harness the resources of the host batch system, regardless of the interface used to submit the virtual jobs.

Note that the scheduling of the virtual batch jobs is at the discretion of the host scheduling policy; these jobs may be interleaved in time and space with jobs submitted by other users. Regardless of whether the host system manages a cycle-scavenging pool or a highly available cluster, the virtual batch jobs may be terminated without notice (*e.g.*, the jobs may be preempted by a higher priority user, the user's allocation may be exhausted, or the execution machine itself may simply fail).

Another practical issue is security. BAD-FS currently uses the Grid Security Infrastructure (GSI) [20], a public key system that delegates authority to remote processes through the use of time-limited proxy certificates. To bootstrap the system, the submitting user must enter a password to unlock the private key at his/her home node and generate a proxy certificate with a user-settable timeout. The proxy certificate is delegated to the remote system and used by the storage servers to authenticate back to the home storage server. This arrangement requires that the user trust the host batch system not to steal the user's secrets, which is a common assumption in such professionally managed "c2c" environments.

4 Experimental Evaluation

In this section, we present an experimental evaluation of BAD-FS under a number of different workloads. We first present our methodology, and then focus on I/O scoping, cooperative caching, explicit cache management, and failure handling, using synthetic workloads to understand system behavior. Finally, we present our initial experience with running real applications on our system.

4.1 Methodology

In the experiments in this section, we build an environment similar to that described in Section 2. We assume the user's input data is stored on a home server; once all jobs have run and all output data is safely stored back at the home server, the workload is considered complete.

We assume that the jobs are run on a distant cluster of machines, accessible from the user's home via a wide-area link. To emulate this scenario, we limit the bandwidth to the home server to 1 MB/s via a simple network delay engine similar to DummyNet [40]. Thus, all I/O between the remotely run jobs and the home server must traverse this slow link. The cluster itself is comprised of a number of Pentium-3 processors with 1 GB of physical memory; each machine in the remote cluster is connected to one another via a 100 Mbit/s Ethernet switch.

To explore the performance of BAD-FS under a range of workload scenarios, we utilize a parameterized synthetic batch-pipelined application. The synthetic workload can be configured to perform varying amounts of endpoint, batch, and pipeline I/O, compute for different lengths of time, and can exhibit different amounts of both batch and pipeline parallelism. As each experiment requires different parameters, we leave those descriptions for figure captions below. However, given previous results in workload analysis, we focus on two particular flavors of workload: "batch intensive", which consists of jobs that exhibit a high degree of batch sharing but little pipeline I/O, and "pipe intensive", which consists of jobs that perform large amounts of pipeline I/O but generate little batch traffic.

Note that in some experiments, the actual parameters we use are artificially small (*e.g.*, a disk cache that is megabytes in size instead of gigabytes). We use such artificial sizes to reduce run times to reasonable values (*i.e.*, such that the jobs complete in hours and not days or weeks). Through further experimentation, we have verified that the results scale to more realistic settings.

4.2 I/O Scoping and Cooperative Caching

In our first experiment, we demonstrate the ability of BAD-FS and the scheduler to use I/O scoping to minimize I/O traffic across the wide area, and the basic effectiveness of cooperative caching. These are both straightforward optimizations, and yet are important. By keeping all pipeline I/O within the remote cluster and aggressively sharing batch data cooperatively across the disks of the cluster, the amount of wide-area I/O traffic is greatly reduced via these simple mechanisms.

The results of our experiments are presented in Figure 5. We use three workloads: "pipe", which is pipe-intensive, "batch", which is batch-intensive, and "mixed", which is in-between. Each workload is run in a number of different system configurations. For example, the lines labeled "Standalone" and "Cooperative" vary whether remote disk caches are managed in a stand-alone or cooperative manner. The lines labeled "Diffused" and "Collocated" vary whether application pipeline data is placed on the same machine where the jobs that produce and consume the pipeline runs. The "Remote" experiment shows values for the situation when a workload is run with all I/O redirected to the home node, as a baseline for comparison. Finally, the "BAD-FS" values represent the workloads running on BAD-FS, which employs both kinds of optimization. Note that in these experiments, we assume copious cache space, and hence capacity-aware scheduling is not employed.

In the leftmost graph of Figure 5, we present the total amount of I/O that is performed across the network. From the graph, we can draw a number of conclusions. Not surprisingly, the caching of batch data in the standalone and cooperative caches greatly reduces batch traffic to the home node; cooperative caching goes a step further by ensuring all but the first reference to a batch data set is retrieved locally. We can also see that the pipeline-oriented optimizations work as expected, transforming pipeline I/O into either LAN traffic ("Diffused") or machine-local traffic ("Collocated"). Finally, the combined effect of keeping all pipeline I/O within the cluster

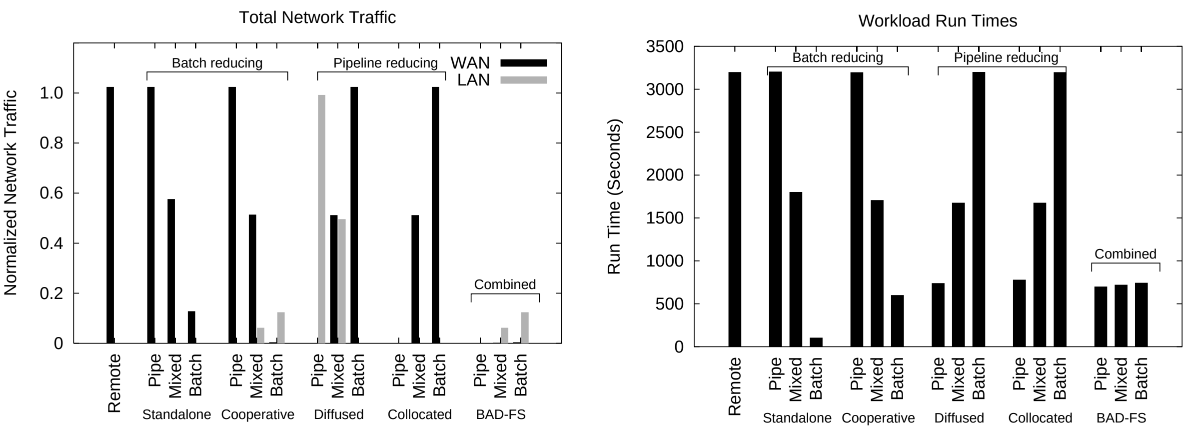

Figure 5: **I/O Scoping and Cooperative Caching: Traffic Reduction and Run Times.** *These graphs shows the total amount of network traffic generated by and run times of a number of different workloads with different optimizations enabled. For this experiment, we run 128 synthetic job pipelines, each with a depth of two for a total of 256 jobs. Across workloads, we vary the relative amounts of batch I/O and pipeline I/O, while holding the amount of endpoint I/O constant. The "pipe" workload generates 10 MB of pipeline I/O and no batch I/O, whereas the "batch" workload generates 10 MB of batch I/O and no pipeline I/O. The remaining component is endpoint I/O, which is comprised of non-shared input data and all final output data. As is common in these types of workloads, amount of endpoint I/O is small (1 KB). The leftmost graph is normalized to the maximum traffic amount transferred, in this case, roughly 2.5 GB of data.*

and aggressively caching batch data cooperatively greatly reduces the amount of wide-area I/O traffic, as seen in the BAD-FS data points.

The rightmost graph in Figure 5 presents the run time of the workloads on our emulated remote cluster. From this graph, we can see that reducing wide-area traffic has a direct impact on run-time; the less traffic that crosses the wide-area, the better the performance. We can also see that the performance difference between collocating pipelines on the machine where processes run ("Collocated") versus placing them anywhere within the cluster ("Diffused") has little effect on final run-time. The reason for this behavior is that local-area network bandwidth is comparable to local disk bandwidth, a configuration which is not uncommon in modern clusters [24]. Thus, given current technology, the BAD-FS scheduler need not be overly concerned with collocated job and pipe placement; simply confining pipe I/O to the cluster is the important optimization.

4.3 Capacity-Aware Scheduling

We next present experiments that show the benefits of explicit cache management in BAD-FS. The previous experiments were run in an environment where remote disk caches were not used to near capacity; with the increasing sizes of data sets in batch workloads and sharing by jobs and users of remote storage, such under-utilization is unlikely to be common in practice. Therefore, the capacity-aware scheduler must carefully manage remote space resources so as to avoid thrashing across the wide-area.

In our first set of experiments, presented in Figure 6, a set of batch-intensive workloads are run. The batch files in this case are large, each taking up some sizable fraction of the available disk cache space in the remote cluster (as varied along the x-axis).

In the leftmost graph, we present the amount of wide-area traffic generated by a number of scheduling policies. Specifically, we compare our capacity-aware scheduler to two simple variants: a *depth-first* scheduler and a *breadth-first* scheduler.

These algorithms are not aware of the data needs of the workflow and base decisions solely on the job structure of the workflow. Depth-first simply assigns a single pipeline to each available CPU, and runs all jobs in the pipeline to completion before starting another. Conversely, breadth-first attempts to execute all jobs in a batch (a horizontal slice of a workflow) to completion before starting the next batch. Each is potentially correct for certain workflows, but each can lead to poor storage allocations.

In this experiment, because the workload is batch-intensive, we expect the capacity-aware scheduling policy to follow the breadth-first approach. As we can see from the figure, this is what happens: the BAD-FS scheduler runs as many jobs as possible that utilize the batch file that currently occupies the cache.

We can make a number of observations from the graph. First, we can see the difference in wide-area traffic when the batch file fits into the cache (*i.e.*, it is less than 100% of the cache size along the x-axis) versus when it does not fit. Once the batch file exceeds the size of the cache, the amount of traffic increases, in this case by a factor of four because there are four sets of eight jobs that access each

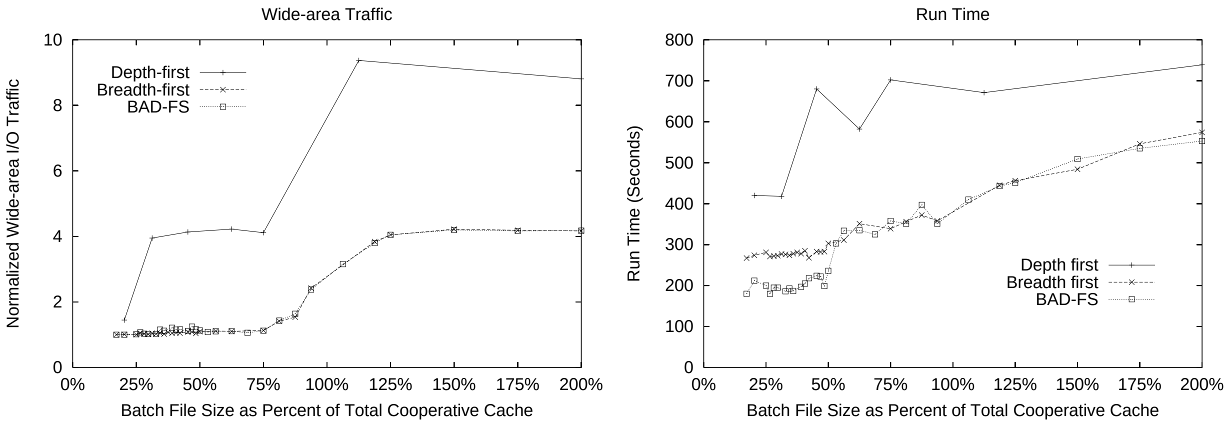

Figure 6: **Batch-intensive Explicit Cache Management: Traffic Reduction and Run Times.** *These graphs the benefits of explicit cache management under a batch-intensive workload. The workload consists of 32 4-stage pipelines; within each stage, each process streams through a shared batch file (i.e., there are 4 batch files total). Batch file size is varied, as a percentage of the total amount of cooperative cache space available across the 8 nodes in the experiment. All other I/O amounts are negligible. Each of 8 nodes has local storage which is used as a portion of the cache. The total cache size available is set to an artificially small value of 16 MB to reduce run time.*

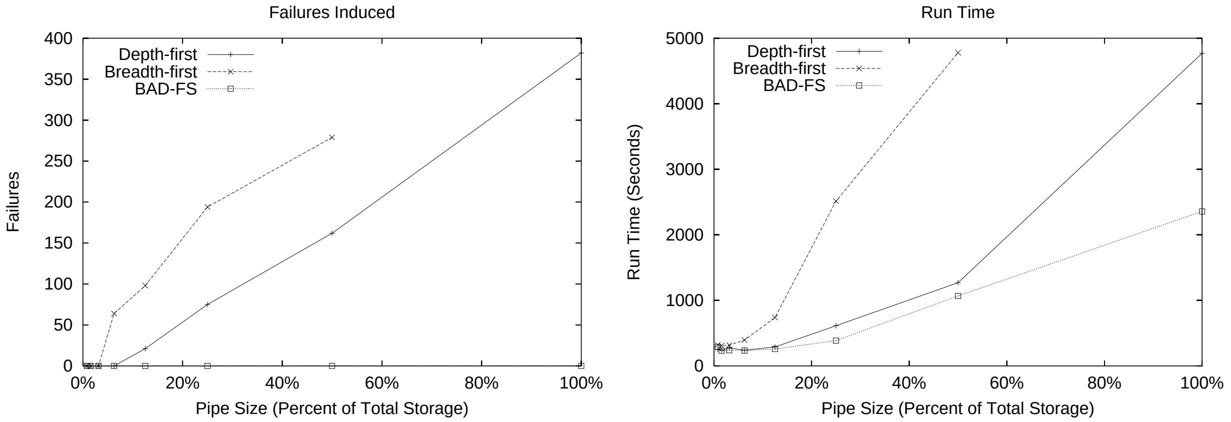

Figure 7: **Pipe-intensive Explicit Cache Management: Failures Induced and Run Times.** *These graphs depict the benefits of explicit cache management under a pipe-intensive workload. The workload consists of 32 3-stage pipelines, and pipe file size is varied as a percent of total storage available. All other I/O amounts are negligible. There are 16 compute servers and 1 storage server in this experiment (representing a set of diskless clients and a single server). The storage space at the server is constrained to an artificially small value of 16 MB to reduce run time.*

batch file. Not surprisingly, we also can see that the depth-first strategy performs quite poorly for batch-intensive workloads, as it does must repeatedly fetch batch files across the wide-area.

We also observe that our cooperative cache management scheme is imperfect; the traffic increase for breadth-first and BAD-FS scheduling arises when the batch file is roughly 80% of the full size of the cache, and not at 100%. The reason for this inefficiency is that the cooperative cache hashing function is not perfectly distributing data across the nodes of the cluster; when the cache nears full utilization, this imperfection overloads some nodes and results in extra traffic to the home server. The implication is that the scheduler must be aware not only of overall utilization of the cooperative cache, but also the utilization of each cache; if any one cache fills, the entire cache should be considered full to avoid overflow.

The rightmost graph of Figure 6 shows the run times for the same set of experiments. As expected, these follow the wide-area traffic numbers closely. One interesting exception occurs at for workloads with smaller batch file sizes, in that the BAD-FS scheduler outperforms the pure breadth-first scheduler. The reason for this improvement is that the pure breadth-first scheduler waits for all processes in one batch to complete before scheduling the next; the BAD-FS scheduler instead will begin the execution of the processes in the next stage of the pipeline if there is room for their data in the cache, thus improving machine utilization and increasing throughput.

In our next set of cache management experiments, we

Induced MTBF	Replication strategy	Pipeline size	
		1 KB	**100 MB**
	always	9.58	8.21
∞	*cost-benefit*	9.64	9.01
	never	9.93	9.33
	always	4.25	3.56
180 sec	*cost-benefit*	5.82	4.85
	never	3.36	2.79

Figure 8: **Failure Handling in BAD-FS.** *The table shows the behavior of the cost-benefit strategy under different failure scenarios, as described in the text. The numbers presented in the rightmost two columns are in jobs per minute.*

focus on a pipeline-intensive workload instead of a batch-intensive one. In this case, we expect the capacity-aware approach to follow the depth-first strategy more closely. Results of this experiment are presented in Figure 7.

In the leftmost graph, we plot the number of failed jobs that each strategy induces. Job failure arises in this workload when there is a shortage of space for pipeline output; in such a scenario, an application that runs out of space for pipeline data aborts and must be rerun at some later time. Hence, the number of job failures due to lack of space is a good indicator of the scheduler's success in scheduling pipeline-intensive jobs under space constraints.

From the graph, we can observe that both of the space-oblivious policies are not able to prevent the system from thrashing. In contrast, the capacity-aware scheduler does not run more pipelines than there is space for, and thus never observes an aborted job. The fruits of this labor are born out in the rightmost graph of Figure 7, where the total run time for the workload is presented. Fewer job failures directly translates to improved throughput.

4.4 Failure Handling

We now show the behavior of BAD-FS under varying failure conditions. Recall that unlike traditional distributed systems, the BAD-FS scheduler has exact knowledge as to how to re-create a lost output file; therefore, whether to make a replica of a file on the remote cluster should depend on the cost of generating the data versus the cost of replication. This need varies with the workload and the system conditions.

Figure 8 shows how the cost-benefit analysis performed by the scheduler suitably adapts to a wide variety of workloads. Shown are four different workloads with extreme ratios of computation to I/O. Each measurement shows peak throughput in jobs per minute, generated from a workload of width 64, depth 3, and jobs of one minute CPU time running on a cluster of 16 nodes. For the lower two cases, an artificial failure generator deleted disks at random with a mean time between failures of 180 seconds, corresponding to the total run time of a single pipe.

In the upper left quadrant, small I/O rates and no failures cause no difference between replication schemes. In the upper right hand corner, replication is not necessary when no failures occur, and approximately a 10 percent throughput penalty is paid to replicate. The cost-benefit strategy, after measuring a single replication, chooses not to do so. In the lower left hand corner, the failure rate makes it highly likely that a single pipeline of three jobs will experience a failure, so replicating intermediate results improves throughput. The cost-benefit case even improves upon always replicating because it only replicates after the second stage; the cost of replicating after the first is an unnecessary expense. A similar story is found in the lower right quadrant.

4.5 Application Experience

We conclude with a demonstration of the system running real applications. Although we have run a number of different scientific batch workloads on our system, due to space considerations, we present results only for one application here, BLAST [4].

In this experiment, we compare BLAST running in four different configurations: *remote*, which redirects all I/O back to the home node across an emulated wide-area link (again, set to 1 MB/s); *standalone*, which emulates AFS-like caching to the home server; *BAD-FS (remote)*, which runs BLAST with BAD-FS on the remote cluster; and *BAD-FS (local)*, whichs runs BLAST on BAD-FS but with the home server accessible across a local-area network link (100 Mbit/s Ethernet). We use 16 machines, and run 64 BLAST jobs in total (recall that each BLAST job is just a single stage pipeline). Each BLAST job takes a small unique input and accesses (in this experiment) a 576 MB shared database.

Not surprisingly, remote throughput is terrible, at 4.67 jobs completed per hour. Standalone caching does better, delivering 18.90 jobs per hour. BAD-FS (remote) with cooperative caching avoids refetching the shared database across the wide area, and achieves 86.36 jobs per hour. Finally, BAD-FS (local) completes 112.93 jobs per hour. Thus, BLAST greatly benefits from the cooperative caching behavior of our system, achieving roughly 76% of "local" performance across a wide-area link.

In our other initial experiences (not shown here), we have found that different workloads benefit from different aspects of our system (*e.g.*, I/O scoping for some pipe-intensive workloads, and capacity-aware scheduling for workloads run upon space-constrained systems). BAD-FS mechanisms thus seem to be flexible enough to support a variety of batch-pipelined workloads, although of course more experience with other application classes is required. In the final version of the paper, we plan to include more of this experience.

5 Related Work

In designing BAD-FS, we drew on related work from a number of distinct areas. Workflow management has historically been the concern of high-level business management problems involving multiple authorities and computer systems in large organizations, such as approval of loans by a bank or customer service actions by a phone company [22]. Our scheduler works at a lower semantic level than such systems; however, it borrows several lessons from such systems, such as the integration of procedural and data elements [43]. The automatic management of dependencies for both performance and fault tolerance is found in a variety of tools [9].

Many other systems have also managed dependencies among jobs. A most basic example is found with the UNIX tool `make`. More sophisticated dependency tracking has been explored in Vahdat and Anderson's work on transparent result caching [51]; in that work, the authors build a tool that tracks process lineage and file dependency automatically. Our workflow description is a static encoding of such knowledge.

The manner in which the scheduler constructs private namespaces for running applications is reminiscent of database views [25]. However, a private namespace is simpler to construct and maintain; views, in contrast, present systems with many implementation challenges, particularly when handling updates to base tables and their propagation into extant materialized views.

There has been much recent work in peer-to-peer storage systems [1, 13, 29, 35, 42, 44]. Although each of these systems provides interesting solutions to the problem domain for which they are intended, each falls short when applied to the context of batch workloads, for the same reasons that distributed file systems are not a good match. However, many of the overlays developed for these environments, such as Chord and Pastry, may be useful for communication between clusters, something we plan to investigate in future work.

Finally, some research in mobile computing bears similarity to our work on BAD-FS. For example, Flinn *et al.* discuss the process of data staging on untrusted surrogates for PDAs and other mobile devices [18]. In many ways, such a surrogate is similar to the BAD-FS storage server; the major difference is that the surrogate is primarily concerned about trust, whereas our servers are primarily concerned with exposing control. Earlier work on Coda also is applicable [28]. Coda uses caching for availability, keeping important files on the local disk of a mobile device so as to avoid unavailability during periods of disconnection. In BAD-FS, servers enact a similar role, caching data so as to avoid downtime when the wide-area link to the home node fails.

6 Conclusions

"He's a big bad wolf in your neighborhood;
not bad meaning bad but bad meaning good."
Run DMC, from 'Peter Piper'

In this paper, we have described BAD-FS, a distributed file system that exposes internal control decisions to an external scheduler. This control, combined with detailed knowledge of workload characteristics, enables the scheduler to carefully manage remote resources and thereby facilitates the execution of I/O intensive batch jobs on clusters across the wide-area.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *OSDI '02*.

[2] R. Agrawal, T. Imielinski, and A. Swami. Database Mining: A Performance Perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

[3] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman. UFO: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, pages 207–233, August 1998.

[4] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, , and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. In *Nucleic Acids Research*, pages 3389–3402, 1997.

[5] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[6] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, and R. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 109–26, Copper Mountain Resort, Colorado, December 1995.

[7] Author list withheld for anonymity. Title withheld for anonymity. In *Conference name withheld for anonymity*, 2003.

[8] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.

[9] Y. Breitbart, A. Deacon, H.-J. Schek, A. P. Sheth, and G. Weikum. Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *SIGMOD Record*, 22(3):23–30, 1993.

[10] J. F. Cantin and M. D. Hill. Cache Performance for Selected SPEC CPU2000 Benchmarks. Computer Architecture News (CAN), September 2001.

[11] S. Chandra, M. Dahlin, B. Richards, R. Y. Wang, T. E. Anderson, and J. R. Larus. Experience with a Language for Writing Coherence Protocols. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.

[12] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[13] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[14] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, Monterey, California, November 1994.

[15] EDA Industry Working Group. The EDA Resource. http://www.eda.org/, 2003.

[16] D. A. Edwards and M. S. McKendry. Exploiting Read-Mostly Workloads in The FileNet File System. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*, pages 58–70, Litchfield Park, Arizona, December 1989.

[17] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 201–212, Copper Mountain Resort, Colorado, December 1995.

[18] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan. Data Staging on Untrusted Surrogates. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[19] B. Ford, M. Hibler, J. Lepreau, P. Tullman, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, Seattle, Washington, October 1996.

[20] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[21] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi- Institutional Grids. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC 10)*, San Francisco, California, August 2001.

[22] D. Georgakopoulos, M. F. Hornick, and A. P. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[23] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing (Lake George), New York, October 2003.

[24] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, California, October 2000.

[25] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.

[26] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[27] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 80–93, Asheville, North Carolina, December 1993.

[28] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[29] J. Kubiatowicz, D. Bindel, P. Eaton, Y. Chen, D. Geels, R. Gummadi, S. Rhea, W. Weimer, C. Wells, H. Weatherspoon, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 190–201, Cambridge, Massachusetts, November 2000.

[30] B. W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP '83)*, pages 33–48, Bretton Woods, New Hampshire, October 1983.

[31] T. L. Lancaster. The Renderman Web Site. http://www.renderman.org/, 2002.

[32] W. Litwin, M.-A. Neimat, and D. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB 20)*, pages 342–353, Santiago, Chile, September 1994.

[33] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of ACM Computer Network Performance Symposium*, pages 104–111, June 1988.

[34] D. Long, A. Muir, and R. Golding. A Longitudinal Survey of Internet Host Reliability. In *Symposium on Reliable Distributed Systems*, pages 2–9, 1995.

[35] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[36] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP '85)*, pages 15–24, Orcas Island, Washington, December 1985.

[37] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. M. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, pages 205–216, San Jose, California, October 1998.

[38] Platform Computing. Improving Business Capacity with Distributed Computing. http://www.platform.com/industry/financial/, 2003.

[39] R. Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, University of Wisconsin-Madison, October 2000.

[40] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[41] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 41–54, San Diego, California, June 2000.

[42] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[43] M. Rusinkiewicz and A. P. Sheth. Specification and execution of transactional workflows. In *Modern Database Systems: The Object Model, Interoperability, and Beyond.*, pages 592–620. 1995.

[44] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[45] R. Sandberg. The Design and Implementation of the Sun Network File System. In *Proceedings of the 1985 USENIX Summer Technical Conference*, pages 119–130, Berkeley, CA, June 1985.

[46] M. Satyanarayanan. A Study of File Sizes and Functional Lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 96–108, Pacific Grove, California, December 1981.

[47] S. Soderbergh. Mac, Lies, and Videotape. www.apple.com/hotnews/articles/2002/04/fullfrontal/, 2002.

[48] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, , and D. Anderson. A New Major SETI Project based on Project Serendip Data and 100,000 Personal Computers. In *Proceedings of the 5th International Conference on Bioastronomy*, 1997.

[49] Sun. Sun ONE Grid Engine Software. http://wwws.sun.com/software/gridware/, 2003.

[50] The PBS Implementation Team. The Portable Batch System. http://www.openpbs.org/, 2002.

[51] A. Vahdat and T. E. Anderson. Transparent Result Caching. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, New Orleans, Louisiana, June 1998.

[52] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 93–109, Kiawah Island Resort, South Carolina, December 1999.

[53] S. Zhou. LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, December 1992.