Computer Sciences Department

The Interaction of Failure and Performance in a Migratory File Service

John Bent
Douglas Thain
Andrea Arpaci-Dusseau
Remzi Arpaci-Dusseau
Miron Livny

Technical Report #1475

May 2003

UNIVERSITY OF
WISCONSIN
MADISON

The Interaction of Failure and Performance in a Migratory File Service

Abstract

We present the design, implementation, and evaluation of a Migratory File Service (MFS), a system designed to exploit semantic knowledge of workloads and user expectations to improve performance and handle failures effectively in wide-area batch scheduling systems. We discuss Hawk, a prototype MFS system which has two novel components: migratory proxies, which cache data at remote clusters, and a workflow manager, which manages the workflow of the system. Hawk integrates aggressive caching and I/O filtering to reduce wide-area traffic, proactively replicates data to avoid regeneration due to failure, and performs fine-grained rollback and recovery to minimize the effort required to recover from failure. Through a case study of data-intensive applications, we demonstrate the benefits of Hawk over traditional approaches, delivering a two to three orders of magnitude increase in performance for jobs that are deployed across a wide-area batch scheduling environment.

1 Introduction

Knowledge is the key to building effective computer systems. Both a sophisticated understanding of the *offered workload* and a solid comprehension of *user expectations* are central to the successful construction of a system that satisfies those who use it. Unfortunately, in many systems, such knowledge is hard to come by; the resulting "curse of generality" has long plagued systems developers, who then must build systems that attempt to satisfy all users at all times under a variety of metrics of success – a difficult if not impossible proposition [7, 19, 61].

If developers could obtain more knowledge of specific workloads and user expectations, they could improve their systems, engineering them to achieve the stated expectations at minimal cost. For example, if a CPU scheduler has reasonable estimates of job completion times, it can perform much more intelligent scheduling than one that does not [45]; if a file server is aware that consistent throughput is what its workload demands, it can be tailored to deliver the required level of performance at low cost [9].

While such knowledge may be difficult to obtain in a general-purpose interactive computing environment, an increasing number of "batch" workloads present systems with an opportunity for enhanced comprehension of workload and user demands. Batch workloads minimally present the system with the set of jobs that need to be run and perhaps some ordering among them. Further, in these environments, user satisfaction is determined directly by throughput of the jobs. While these batch workloads were formerly relegated to a few specialized scientific workloads [6], they now are common across a broad range of important and often commercially viable application domains, including genomics [3], video production [62], simulation [11], document processing [18], data mining [2], electronic design automation [17], financial services [49], and graphics rendering [38].

Batch workloads are typically run in controlled local-area cluster environments [41, 67]. However, organizations that have large workload demands increasingly need ways to share resources across the wide area, to lower costs and increase productivity. An obvious solution is to simply run an existing batch scheduling system across machines in the wide area; like many obvious solutions, this approach is fraught with difficulty.

The prime limitation in running one of the many extant batch scheduling systems across the wide-area is found in how they typically handle the input/output demands of applications. These CPU-centric systems treat I/O as a second class citizen, often redirecting all of the input and output of jobs back to the home node of the user [24, 40]. As data demands grow, such a "remote I/O" approach severely limits the types of jobs these systems can effectively process.

To address these problems and make I/O a first class citizen in a wide-area CPU sharing environment, a new file service paradigm is required. Such a service should take a holistic view of applications and the system, taking into account available knowledge of applications to improve performance. Such a service must handle the failures that are common in these environments in an efficient manner. Finally, such a service must manage itself, as there is no system manager to ensure that jobs at the remote site are running smoothly.

In this paper, we introduce a service that meets the aforementioned criteria. We call such a system a *migratory file service*, as it migrates the I/O environment of a user or application to the remote CPU site. A migratory file service relies upon a single primitive that is common to all CPU sharing environments, namely the ability to

launch a job on a remote site, to dynamically instantiate a *virtual batch system overlay* in the remote execution environment. The virtual batch system is used to create an automatically managed computational and storage environment for the remotely run jobs, which is tailored on-the-fly to the demands of the running applications.

We describe our prototype migratory file service known as Hawk, which is designed to take advantage of workload knowledge in two primary ways. First, Hawk improves *performance* through the use of self-organizing peer-to-peer *migratory proxies*. These proxies first classify and then treat the different types of I/O common in batch workloads accordingly. Hawk proxies aggressively cache input data cooperatively [16, 20] across remote nodes so as to reduce wide-area communication and improve file access latency. Further, Hawk proxies carefully filter output traffic from remotely-run processes, migrating only the needed files back to the home file server of the user. Both knowledge-driven optimizations serve to reduce wide-area network transfers substantially and enable the deployment of data-intensive applications.

Second, Hawk gracefully handles *failures* through an intelligent *workflow manager*. Both migratory proxies and jobs can fail at the remote site. The workflow manager uses logging and transactional techniques [27] to perform fine-grained rollback and recovery, ensuring progress despite sporadic availability of remote resources. Further, when a migratory proxy fails, it may hold "dirty" data from a job that recently completed, *i.e.*, data that has not yet been committed to the home node of the user. Whereas data loss is viewed as catastrophic in typical file services, the workflow manager understands how to regenerate a given file (*i.e.*, by re-running the job that generated the file). In addition, if a file is costly to regenerate, Hawk replicates the file proactively, reducing the likelihood that the job needs to be run again and improving overall throughput.

To achieve high throughput for batch workloads in failure-prone environments, Hawk does not view failure and performance in isolation. Rather, Hawk takes an end-to-end perspective [57], understanding that the true measure of success in batch systems is delivered job throughput. By combining knowledge of the workload (*e.g.*, which files are important) with knowledge of user expectations (*e.g.*, high throughput for jobs), Hawk handles performance and failure in an integrated and unified manner.

We study Hawk through controlled microbenchmarks as well as by running a suite of demanding scientific applications. Through microbenchmarks, we demonstrate that caching of input data is crucial for performance, and that localization of ephemeral I/O greatly reduces wide-area I/O traffic. We also demonstrate how Hawk reacts to failure with fine-grained rollback and recovery, and the need for replication of output data that is crucial to performance in some scenarios. Finally, through an application study of data-intensive scientific codes, we show that Hawk improves the throughput of these applications by two to three orders of magnitude over traditional approaches.

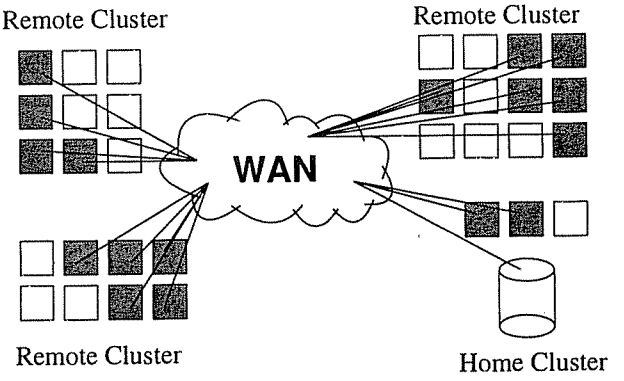

Figure 1: **The Target Environment.** *The figure presents a depiction of a typical CPU sharing environment. The user has their important data stored on a file server in their home cluster, and is running jobs on machines at geographically-dispersed clusters. Each box represents a machine, and shaded boxes are machines upon which the user is currently running jobs.*

The rest of this paper is structured as follows. We describe background in Section 2, and give an overview of Hawk in Section 3. Then, in Sections 4 and 5, we describe the performance optimizing and failure handling of Hawk. We present our application study in Section 6, related work in Section 7, and conclude in Section 8.

2 Background

In this section, we describe the setting for migratory file services. We first discuss the environment where such a service will likely be deployed, and then present an example from the perspective of a user. We conclude with a discussion of the expected structure of batch workloads.

2.1 Environment

Although wide-area sharing of untrusted and arbitrary personal computers is certainly a possible platform for batch workloads [63], we believe that the primary platform for these types of throughput intensive workloads will be 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. Figure 1 presents a typical environment.

We further assume that each cluster exports their resources through some type of CPU sharing system, and that each site that participates has local autonomy over their resources. Although a user may be able to use remote resources at a given time, they 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 simply a prioritized preemption by the owning site.

We sometimes refer to this more organized, less hostile, and well managed collection of clusters as *c2c (cluster-to-cluster)* computing, in obvious 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; thus 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 directly applicable to the c2c domain; indeed, our migratory proxies self-organize and manage themselves in a manner reminiscent of peer-to-peer file systems [37, 56].

2.2 Example Usage

We now consider a user who wishes to run a data-intensive, high-throughput workload. After the user has developed and debugged the application on their home system, they are ready to run hundreds or thousands instances of their application on all available computing resources, using a remote batch execution system such as Condor [40], LSF [67], PBS [65], or Grid Engine [64].

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 NFS 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 their workload. The first option is to simply submit their workload to the remote batch system. With this option, 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, wide-area network bandwidth and latency limitations is not sufficient to handle simultaneous requests from many data-intensive applications running in parallel. Second, all I/O from the application 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 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 many users of such systems go to these lengths simply to run their jobs!).

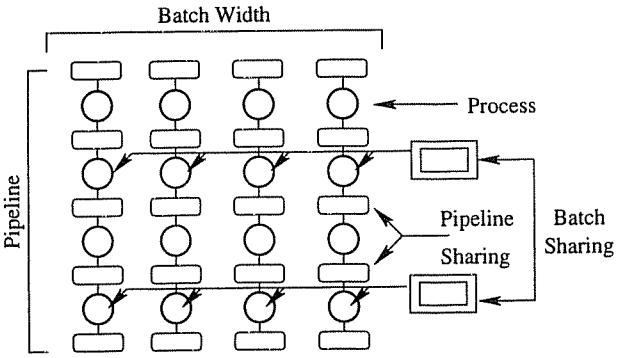

Figure 2: **A Batch-Pipelined Workload.** *A typical batch-pipelined workload is depicted. A single pipeline represents the logical work that a user wishes to complete, and is comprised of a series of processes. Users often assemble many such pipelines into a batch to explore variations on input parameters or other input data.*

A migratory file service solves these problems by creating a personal data-intensive computing environment on the basic substrate of one or more batch systems. The MFS is responsible for deploying a task force of services that identify the combined compute and storage resources in a cluster and export them in manner that hides the underlying complexity and faulty behavior. We discuss the details of Hawk, our prototype MFS, in Section 3.

2.3 Workloads

We now define the expected data-intensive, high-throughput workload in more detail. As illustrated in Figure 2, these workloads are composed of multiple independent pipelines; each pipeline contains sequential processes that communicate with the preceding and succeeding processes via private data files. Thus, we refer to this class of workloads as *batch-pipelined*. 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 faster or slower than its siblings.

The key difference between a single application and that of a batch-pipelined workload is the 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 the workloads, by breaking I/O activity into three categories: *endpoint*, which represents the 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.

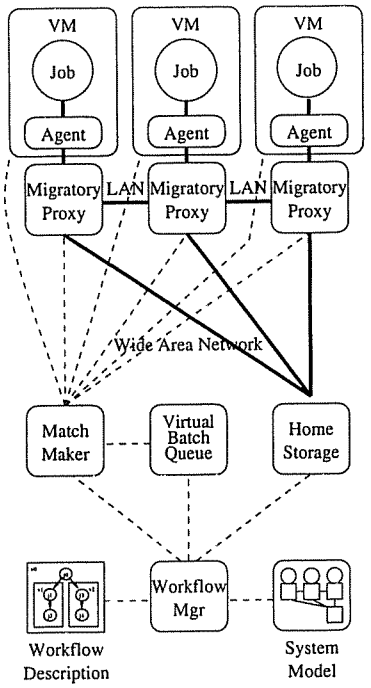

Figure 3: **The Hawk Migratory File Service.** *The components of the Hawk MFS are presented. A workflow manager resides at the user's home and manages execution based on the workflow description. Remote proxies serve and cache data requests for jobs running on remote nodes. Job I/O requests are redirected by an interposition agent to the proxies.*

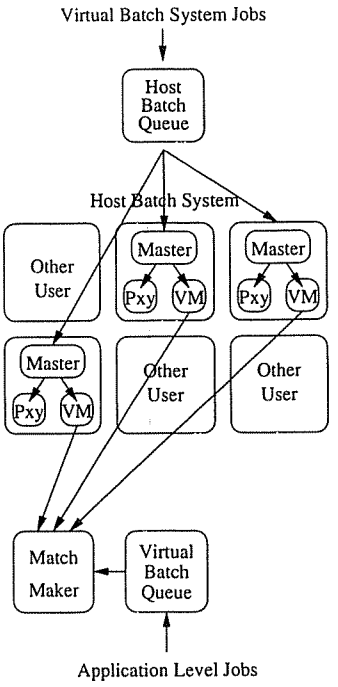

Figure 4: **Deploying a Virtual Batch System.** *The figure depicts a virtual batch overlay. Six nodes are in the cluster of interest, three of which have been assigned to the user. Hawk uses the ability to launch a job on these nodes to instantiate it's own virtual batch system, including migratory proxies to handle I/O and a VM to run it's jobs.*

3 Hawk Overview

In this section, we present the key components of Hawk, our implementation of a migratory file service. Hawk combines the advantages of a distributed file system with those of traditional batch systems to meet the semantic and performance needs of high throughput applications.

An overview of Hawk is shown in Figure 3. Both data traffic (solid lines) and information and control (dashed lines) are shown. From this figure, we see that the nodes in the remote cluster run both user applications and migratory proxies, whereas the home storage server for these applications is located across the wide area.

Our description of Hawk focuses on three major components. First, we describe how Hawk can be deployed as a virtual batch system on top of an autonomous remote system which exports an arbitrary batch execution environment. Second, we give an overview of the mechanisms required to efficiently support I/O in a remote setting, focusing primarily on the Hawk migratory proxies. The role of the migratory proxies is to aggressively cache and share data in a cooperative fashion, avoiding as many costly interactions with the user's home storage server as possible. Third, we discuss the Hawk workflow manager. The role of the workflow manager is to track the dependencies across jobs and data so that the only essential output data is sent back to the user's home, to perform fine-grained rollback and recovery transparently in the face of job or proxy failure, and to replicate output data that would be costly to otherwise regenerate.

At the end of the section, we present an argument as to why a more traditional distributed file service is not the correct solution for this type of workload and environment, and discuss other issues that are germane to the design and implementation of a migratory file service.

3.1 The Virtual Batch System

Hawk must be able to run on autonomous remote sites, with no intervention required by administrators or users. The implication is that Hawk cannot assume that it has any particular software packages installed or special privileges on the remote site. The only assumption made by Hawk is that the remote site runs a simple *host batch systems*, such as Condor, LSF, PBS, or Sun's Grid Engine. Because these systems differ in the services they offer, Hawk assumes only the most basic functionality: the ability of the host batch system to queue, run, and (if needed) halt running executables.

To establish the necessary control over the remote site, Hawk does not use the host batch system directly, but instead establishes a *virtual batch system* as an overlay across the host systems; this approach is similar in spirit

to that which has been applied to virtual machines [22] and networks [4]. With this approach, *glide-in jobs* are submitted to the host batch system [25]; the glide-in jobs are not user applications, but a package of executables and configuration files to instantiate the virtual batch system. Hawk currently employs Condor [40] as its virtual batch system, but could use any fault-tolerant, opportunistic distributed system in its place. By dynamically instantiating a virtual batch system on top of the extant physical batch, Hawk ensures that it has the control over remote resources that it requires.

Note that the scheduling of the glide-in 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 glide-in 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). The host system may restart the glide-in job or it may not; Hawk handles both cases by keeping a supply of virtual batch jobs in the host queue.

Figure 4 depicts how the virtual batch system is built. The glide-in jobs submitted to the host batch queue are a package of three executables: a master, virtual machine, and migratory proxy. The *master* process completes the configuration and starts the other two processes. The *virtual machine*, a direct component of Condor, exports the resources of the machine. This process accepts and executes program binaries, periodically reporting back vital statistics about the execution, and at the completion of an application, reports the final state, transfers output files, and cleans up resources. Finally, the migratory proxy exports the storage resources found on that machine to the jobs running on the cluster; this important component of Hawk is described in more detail in the next section.

There are two additional components within Condor that are used by Hawk: the matchmaker and the user's virtual batch queue. The *matchmaker* is a dynamic catalog of the participants in the system, introducing compatible parties to each other. The ClassAd [50] resource description language is used to accept advertisements from the virtual machines and proxy caches as well as to specify that jobs prefer or require machines with certain qualities, such as an amount of memory or particular CPU type. The *virtual batch queue* persistently stores a user's jobs; if the queue crashes, any running jobs fail, but the queue state will be recovered and failed jobs will be restarted automatically.

3.2 The I/O Subsystem

The I/O portion of Hawk has the core responsibility of virtualizing the storage resources of the remote site. Through these components, Hawk implements the essential mechanisms for providing transparent access to private namespaces, caching data files to improve performance, and the underlying mechanisms for dealing with storage failure. Figure 5 presents the interaction of the I/O components.

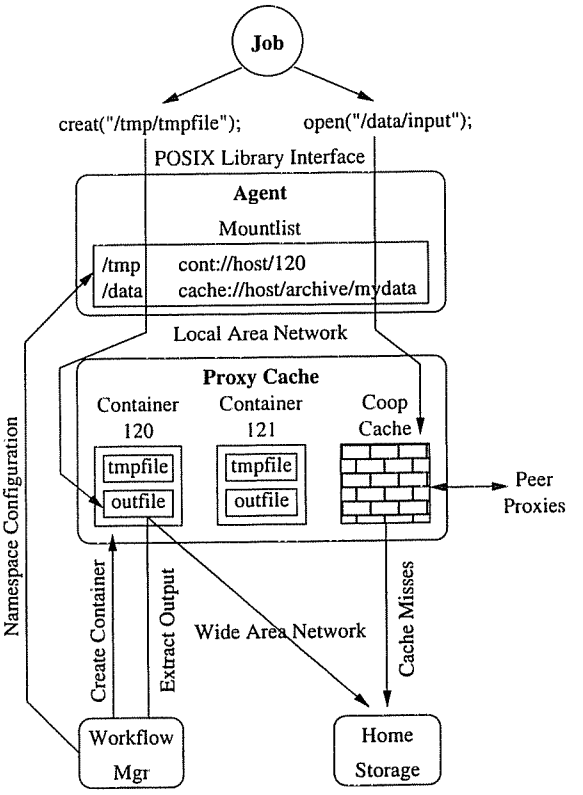

Figure 5: **Interaction of the Agent and Proxy.** *In the figure, a job opens two files that are a part of its namespace. I/O requests are redirected by an interposition agent to the local proxy, which may either directly service the request, request the given data from a peer proxy, or fetch the data from the home server.*

3.2.1 Migratory Proxies

The most important component of the Hawk I/O infrastructure is the migratory proxy. Each proxy exports the storage resources of the node upon which it runs. Migratory proxies provide two important abstractions: the *cooperative cache* and *leased containers*.

The cooperative cache is a global read-only cache managed by the group of proxies. The portion of the cooperative cache within each proxy is a demand-paged cache with LRU replacement. Unlike previous cooperating caching schemes that manage cluster memory in a global fashion [16, 20], the Hawk cooperative cache stores data in the local *disks* of each remote node, thus avoiding costly fetches from the home storage server.

Upon a request from a client, the proxy fetches the requested data blocks, preferably from its peer proxies but from the home server if necessary. Several variations on cooperative caching are possible [16]; the Hawk algorithm works as follows. Each proxy in a group is responsi-

ble for an equal share of the data available to clients. Upon request from a client, a proxy checks to see if it contains the needed data. If it does not, the proxy applies a hash function to the filename and block number, yielding a linear hash index. This map transforms the index into the name of a peer proxy, which is then queried for the data. If the peer has the data, it is returned; otherwise the peer performs a read request to the archive. In the future, it may be interesting to investigate whether more sophisticated hashing schemes could be applied [47]; however, as we will see in Section 4, this simple hash-based scheme achieves much of the possible benefit.

In order to cooperate, proxies must agree on their organization. To do so, each proxy first bootstraps itself into a group by contacting the archive server, registering itself as a proxy, and requesting the list of available proxies; other techniques, such as expanding ring multicasts or the use of a known global data file, could be used as well. Proxies then self organize to select their *coordinator* by using the eldest proxy known to the archive server. Once the coordinator is selected, each proxy contacts the coordinator, which maintains the list of current peers and assigns each a position in the hash map. New proxies register with this coordinator, who monitors their health and pushes new hash maps for each join and leave operation. If the coordinator fails, the proxies fall back to the bootstrap protocol with the archive server.

The second abstraction implemented by the proxies is the *leased container*. A leased container is scratch space used for storing application output and other temporary files. Containers are created by the workflow manager and are cleaned up either by the workflow manager when finished or by the migratory proxy when the lease expires. A transactional interface is used to name and create containers, which simplifies recovery in the event of failure.

3.2.2 Interposition

The remaining portion of the Hawk I/O subsystem provides connectivity between clients and migratory proxies. Specifically, user jobs connect to nearby proxy caches by using an *interposition agent* [34]. Our interposition agent inserts a library into an unmodified executable to redirect I/O operations to their proper destination; thus, no source code modifications are required. Upon I/O operations, the agent maps the pathnames used by the application into the physical names used by the proxies. This is performed by consulting a *mount list*, programmed by the *workflow manager* as described below; in this manner, each instance of the application can transparently operate in its own private namespace.

Interposition agents and leased containers combine allow the I/O system of Hawk to cleanly differentiate failures from errors. When an agent contacts a proxy, it is able to tell the difference between a missing container and a missing file in a valid container. The former indicates a failure of some kind (*e.g.*, a crashed proxy, an expired lease, or a deliberate cleanup). In this case, the agent forces the process to exit with a signal indicating an I/O error; this signal propagates back to the workflow manager, which must then recover from the failure. The latter indicates that the job is looking for a file that does not exist and should be passed a "file not found" error.

```
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 v1
  ftp://home/mydata
volume v2 scratch
volume v3 scratch
mount v1 a /mydata
mount v1 c /mydata
mount v2 a /tmp
mount v2 b /tmp
mount v3 c /tmp
mount v3 d /tmp
extract v2 x
  ftp://home/out.1
extract v3 x
  ftp://home/out.2
```

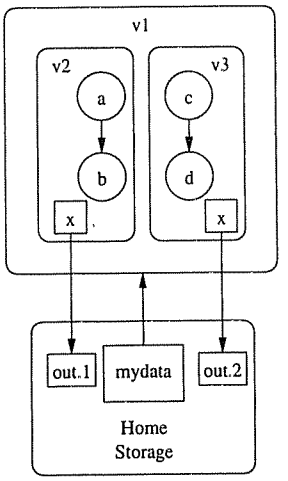

Figure 6: **Workflow Language and Schematic.** *An example workflow is depicted. A directed graph of jobs is constructed via the* `job` *and* `parent` *keywords, and the file system namespace presented to jobs is configured via* `volume` *and* `mount` *directives. The* `extract` *keyword indicates which volumes must be committed to stable storage at the home storage server upon job completion.*

3.3 Workflow Manager

The final task that Hawk must handle is to manage the flow of batch execution, incorporating knowledge of the user's workload to improve performance and recover cleanly from failures. Two types of information are useful. First, information about which files are temporary allows Hawk to avoid copying output files over the wide-area to archival storage. Second, information about which jobs produce or consume each files allows Hawk to recreate output which is lost due to a node failure.

In Hawk, these responsibilities are that of the *workflow manager*. The manager accepts a *workflow* description of a large set of work to be done, discovers storage and execution resources on which to carry it out, and then executes the plan, taking failures into account.

3.3.1 Workflow Description

Figure 6 shows an example and a schematic rendering of the 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 proxy cache; 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.

We note that users running a large number of interdependent jobs must always express these types of dependencies. To successfully execute thousands of jobs, 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 that those operations are performed, much in the way that relational queries separate what the user wants from how it gets computed [13]. This abstraction allows users to be blissfully unaware low-level system details (*e.g.*, how failures are handled), while giving the system powerful information about jobs and data.

3.3.2 Basic Operation

The workflow manager 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 workflow manager refreshes is model of the system by querying the matchmaker for a list of resources.

With the information expressed in the workflow, the task of determining which output files are temporary and can be cached by the proxies locally is straightforward. By default, all files are kept only within the Hawk proxies; only when a file is explicitly extracted is the output file sent to archival storage.

3.3.3 Handling Failures

A key component of the workflow manager is found in how it makes Hawk robust to failures. The workflow manager can handle failures of the matchmaker, job queue, storage containers, and the manager 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 manager fails, it recovers from the log and resumes operation without losing jobs or storage containers.

Hawk handles jobs and storage failures by waiting for passive indications, and then conducting active probes as necessary. For example, if a job returns to the workflow manager with an abnormal exit code indicating an I/O failure (generated by the interposition agent), it suspects that the proxy servers housing one or more of the containers assigned to the job is faulty. The manager then probes the proxy caches 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 workflow manager deletes the container 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, Hawk needs to restart the jobs that created the lost files needed by the stopped jobs. To determine this set of "creation processes", the manager durably records the file dependencies that occur during execution. Of course, these restarts may be recursive; the creation processes may in turn rely upon input data that has been lost and must be recreated as well.

In order to avoid these expensive restarts of a workflow, Hawk supports replication of volumes across proxies. 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, Hawk performs a cost-benefit analysis at run-time to determine when a volume should be replicated.

Note that a Hawk does not handle permanent failures of the home storage server. This non-trivial but well-understood problem can be managed through backup [33] and replication [8, 48]. Temporary disconnection to the home storage node however can be tolerated and progress can continue as long as the necessary input data can be found in the migratory file service and there is sufficient capacity for the outputs.

3.3.4 Workflow Management and Transactions

In many ways, the workflow manager draws on techniques from the field of database management, and in particular, transaction processing and ACID properties [27]. A pipeline can be thought of as a *transaction*, and each job within a pipeline as an *atomic* operation. Each namespace

provides *isolation* among the processes of a workflow, to the degree required by the application. *Consistency* is managed according to application desires by controlling execution in the directed graph of computation; if a job should not access a file until another job is finished writing it, the exact dependency should be specified in the workflow. Finally, when a file is extracted from a volume and committed to the home server, the results of the job are thus made *durable*.

3.4 Why Not A Distributed File System?

One might initially assume that data-intensive applications in a distributed environment are well served by a distributed file system (DFS) [60]. A DFS is a well-understood abstraction providing a coherent namespace and uniform data accessibility (modulo permissions) across a set of physically distributed machines. However, a DFS is not appropriate for personal wide-area data access in the presence of failures. Beyond the non-trivial technical and social barriers of deploying a DFS across geographically-dispersed clusters of machines, there is an underlying structural reason:

> *Distributed file systems provide an unnecessarily strong abstraction that is expensive to provide in a fault-prone, wide-area environment.*

A migratory file service differs from a traditional distributed file system in the following important respects:

Namespaces. Traditional file systems provide participants with access to the entire namespace implemented by the system (modulo permissions), and allow for all to consistently read and write the same data (by some definition of consistency). In contrast, an MFS allows each participant to construct its own private namespace, with both global and local components. Only those components needed by a job are stitched into its namespace.

Consistency. Much of what complicates distributed file system design and implementation revolves around cache consistency [5, 12, 44]. In an MFS, in contrast, the responsibilities of coherence and consistency are specified in the workflow description, which greatly reduces implementation complexity while providing the expected benefits of caching.

Selective commit. Traditional file systems assume that all newly created data has high value and must be forced to stable storage at specific times [31, 42] or at least within small bounded intervals [53, 58]. In contrast, an MFS recognizes that the vast majority of data is ephemeral and expensive to move, and thus accounts for the different types of I/O in its design. By doing so, an MFS can optimize data movement aggressively and thus achieve excellent performance via workload knowledge.

Fault tolerance. Traditional file systems have limited recourse for dealing with failed clients. Those that enforce immediate commit of newly-written data will not lose data, but also cannot continue to operate in the face of failures. Those that permit delayed write-back can operate through failures, but then must appeal to system administrators for reconciliation [36], or simply accept the possibility of inconsistent data [35]. Through the use of private namespaces and selective commit, an MFS is able to understand the overall structure of a workload and determine the precise consequences of a failed compute or storage element. This allows an MFS to reap the performance and availability benefits of delayed write-back while maintaining transparent failure recovery.

3.5 Other Issues

One of the main other issues that needs to be addressed in any wide-area system is that of security. Hawk currently uses the Grid Security Infrastructure (GSI) [23] for all of its interactions, GSI is 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. This proxy certificate is transported to the remote batch system along with the glide-in job and serves as the credential for the proxies and virtual machines. As the components communicate with each other, the home storage server, and other components at the user's home site, they perform user-to-user authentication and present themselves as the submitting user in the limited role of "proxy."

This system requires that the user trust the host batch system to protect the proxy credentials from exposure while in transit to the system, while idle in the batch queue, and while in use on an execution node. If stolen, the credentials can be used to impersonate the user in the role of proxy. This danger is not particular to Hawk, but is present in any system where delegation of credentials is performed. If the user does not place a great degree of trust in the host system (although we believe that they commonly might), the power of the credential can be limited to bound the damage from a stolen proxy, by either limiting its lifetime, delegating a proxy role which is only authorized to perform the tasks necessary to the workload, or limiting authorization of the credential to a certain set of known hosts.

4 Exploiting Knowledge for Performance

In this section, we explore how the Hawk migratory file service exploits its knowledge of the workload to optimize job throughput. In this set of experiments, we assume an idealized system in which no failures occur and show that Hawk aggressively caches and filters I/O to reduce communication and improve performance. In the next section, we assume (realistically) that failures are common

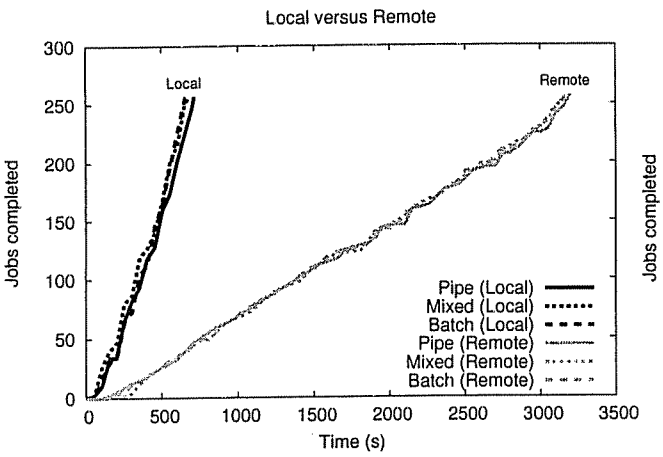

Figure 7: **The Costs of Remote I/O.** *The experiment measures job completion time for the three synthetic workloads, run either in a local cluster or in a remote cluster across an emulated wide-area link.*

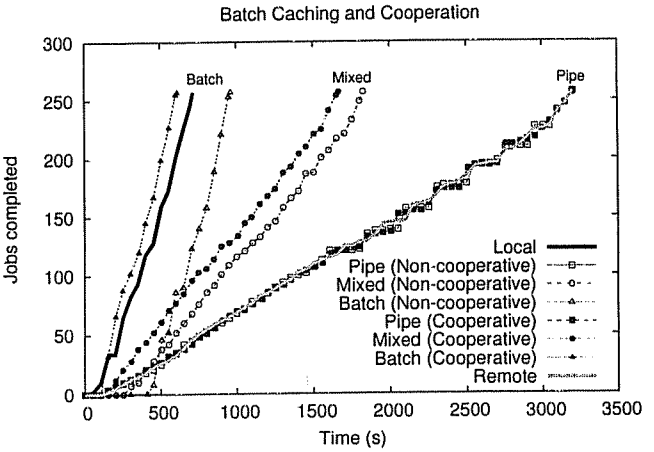

Figure 8: **Caching Batch I/O.** *This experiment measures job completion time when batch I/O caching is utilized. Performance with both non-cooperative and cooperative proxies is shown.*

and demonstrate how Hawk integrates performance and failure management to improve job throughput.

4.1 Experimental Setup

In the experiments in this and the next section, we assume the user's input data is stored on a home file server. After all jobs have run and all output data is safely stored back at the home file server, the workload is complete. The metric of success is simply job throughput.

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 file server to 800 KB/s via a simple network delay engine similar to DummyNet [52]. All I/O between the remotely run jobs and the home file server must traverse this slow link. The cluster itself is comprised of 32 550-MHz PIII Katmai processors with 1 GB of physical memory; each machine is connected to one another via a 100 Mbit/s Ethernet switch.

To explore the performance of Hawk under a range of workload scenarios, we utilize a parameterized synthetic batch application. In each scenario, we measure the job completion rate of a synthetic workload with 128 pipelines, each with a depth of two jobs for a total of 256 jobs. Across scenarios, we vary the relative amounts of batch I/O and pipeline I/O, while holding the amount of endpoint I/O constant. Recall that batch input is shared across multiple pipelines, whereas pipeline I/O is temporary and used for communication within a single pipeline. 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 workloads, the amount of endpoint I/O is small; for all experiments below, we set endpoint I/O to 1 KB of input and 1 KB of output per pipeline.

4.2 Local versus Remote

We first establish the baseline performance of running jobs remotely versus in an idealized local setting (*i.e.*, one in which the home file server is a locally-accessible machine). Results of our experiment are shown in Figure 7. For both the local and the remote case, we present the results from three different workloads: *batch intensive*, in which the amount of batch I/O is set to 10 MB and there is no pipeline I/O, *pipeline intensive*, in which there is 10 MB of pipeline I/O and no batch I/O, and *mixed*, with 5 MB of both batch and pipeline I/O.

In this and subsequent graphs, we increase time along the x-axis and plot the cumulative number of jobs completed along the y-axis. The slope thus represents the throughput. From the graph, we observe the high cost of performing I/O across wide-area links. In the remote case, all input and output traffic travels across the slow link, and thus performance suffers dramatically, with jobs finishing roughly six times slower than in the local setting. Because the naive remote system does not discriminate across the types of I/O, the only factor that influences performance is the bandwidth to the home file server.

4.3 Caching Batch Input

Having established a baseline for both best-case (local) and worst-case (remote) performance, we now apply our semantic understanding of the I/O characteristics to improve performance while running in a remote environment. We concentrate first on batch I/O.

Figure 8 shows how Hawk performs when caching batch data in migratory proxies. With batch caching enabled, remotely-run jobs can approach "local" performance levels for batch-intensive workloads. We examine both individual caching by each migratory proxy and cooperative self-organized caching across all migratory

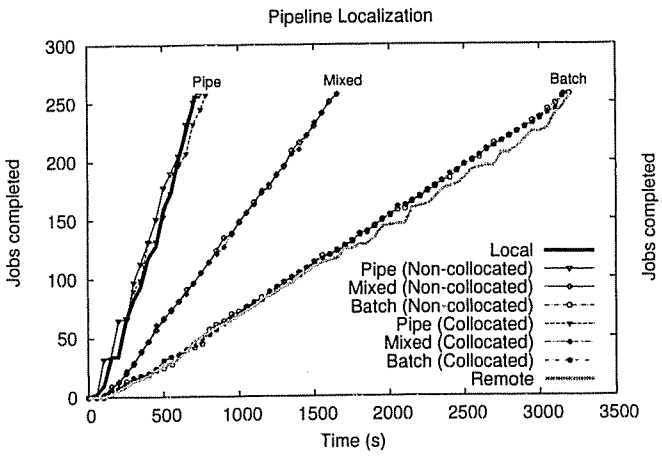

Figure 9: **Localizing Pipeline I/O.** *This experiment measures the performance of Hawk when knowledge of the workload is applied in order to localize pipeline I/O. In the experiments labeled "collocated", the workflow manager places jobs and pipeline data on the same node.*

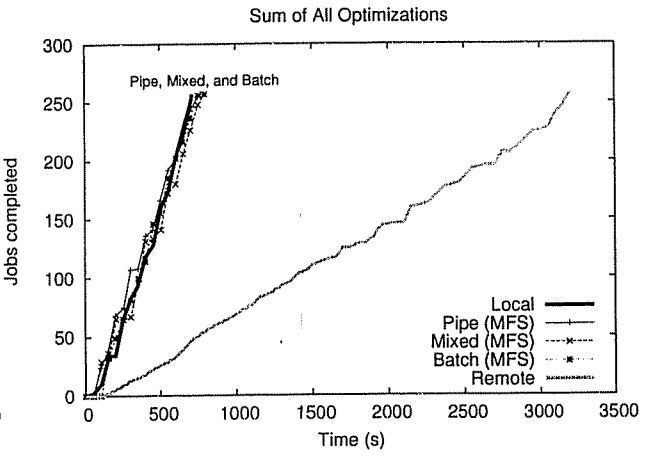

Figure 10: **Putting It All Together.** *This experiment shows how Hawk performs at local rates regardless of the ratio between batch and pipeline I/O, by localizing pipeline I/O and cooperatively caching batch data. The pipeline localization utilizes the collocation variant.*

proxies. Because each migratory proxy has enough capacity to completely cache the batch data, the rate of job completion for each approach is similar. The only difference between the two approaches occurs during initialization: cooperative migratory proxies share batch data and thus avoid multiple trips to fetch the same data from the home node, whereas the individual (non-cooperative) migratory proxies contend for wide-area bandwidth and the home file server and are therefore not able to to fetch the batch data as quickly.

The cooperative batch caches can even outperform local job execution. With a cache proxy running on every node of the cluster, each job has a dedicated proxy; due to this parallelism, each job obtains a higher I/O bandwidth than when each job contends for the same server.

4.4 Localizing Pipeline I/O

We now turn our attention to pipeline I/O. All pipeline I/O is private per pipeline; thus, if Hawk has knowledge of its ephemeral nature, it can avoid moving any of the private pipeline I/O across the wide area. Further, if the workflow manager can collocate jobs of a pipeline with their pipeline inputs and outputs, the local-area traffic in the cluster can also be reduced.

As one can see in Figure 9, when Hawk keeps all pipeline I/O in the cluster (and thus avoids the WAN link), both the pipeline-intensive and mixed workloads benefit, the former more than the latter. The batch workload does no better with this optimization since it contains no pipeline I/O. The figure also shows that the difference between the collocated and non-collocated approaches is quite small; with copious bandwidth available in the remote cluster's LAN, the coordinated placement of pipeline data and jobs is not important.

4.5 Putting It All Together

Finally, we demonstrate how the combination of cooperatively caching batch data and localizing pipeline I/O in the migratory proxies transforms a remote cluster into a "local" system from the standpoint of performance. Figure 10 runs the same synthetic workloads, with all optimizations enabled. The graph shows that regardless of the job type (batch-intensive, pipeline-intensive, or mixed), performance is excellent, quite similar to the situation where all of the jobs are run locally, and sometimes better. As shown in Figure 11, the key to the success of Hawk is that it reduces the amount of network communication, particularly to the home file server.

5 Exploiting Knowledge to Handle Failure

In this section, we explore how Hawk employs high-level workload knowledge to recover from failures without operator intervention.

Hawk requires a transactional interface to the batch queue and the proxy caches, as well as a persistent log to record remote actions and recover from failures. In the case of the job queue, a `begin` message is used to request a new unique job ID. This ID is immediately recorded in the manager's log for crash recovery. The manager provides the details of the job, and then logs and issues a `commit` to release it for execution. When the job completes, the queue informs the manager, which may then extract the necessary details such as the exit code and resource consumption statistics. Once satisfied, the manager may issue a `delete` to remove the record of the job. A similar discussion applies to the creation and management of leased containers.

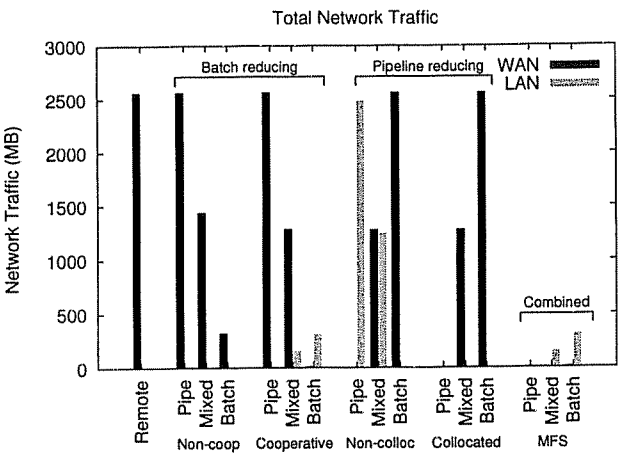

Figure 11: **Network Traffic Summary.** *This bar graph confirms that the performance improvements of the various schemes is due to a corresponding total amount of network traffic incurred by the home file server.*

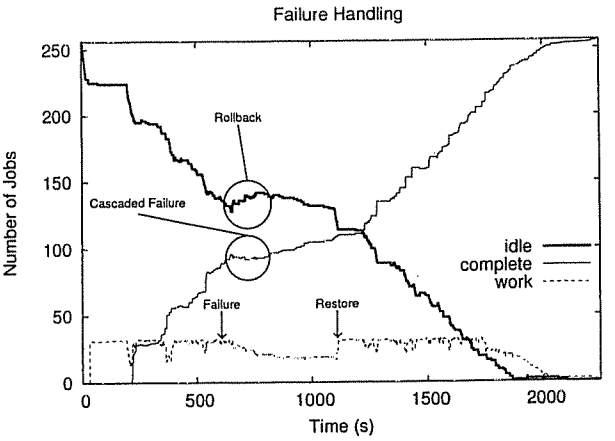

Figure 12: **Failure Timeline.** *This figure is a trace of Hawk executing a workload in a faulty environment. 15 machines were removed from the cluster at about 700 seconds, and replaced at about 1200 seconds.*

The workflow manager is not immediately aware of most external failures in the system. Although it would be possible to continuously probe the job queue and proxy caches for their status, this task could quickly grow to occupy the manager's time as well as waste resources useful to others. Instead, the manager waits for passive indications of failure, and then performs selected active probes to determine the extent of the damage. There is a key interaction between the failure of jobs and volumes that must be observed. If a job fails, the volumes it depends upon must be verified. If any of these are corrupt, any jobs that produced data they contain must also be rolled back, even if they have already completed.

Figure 12 demonstrates the behavior of Hawk in a faulty environment. The figure is a timeline of a workflow executing on a cluster in which an outage of 15 machines was induced about 700 seconds into the run. Such failures happen in real clusters for a variety of reasons, including preemption by other users, scheduled maintenance or software installation, or correlated failures of software or hardware.

After the failure, the number of jobs in the "working" state declines slowly as the workflow manager passively discovers the failure, and then actively probes the system to see what proxy caches have failed. This has two effects on the running workload. First, running jobs that were directly affected by the failure roll back to an "idle" state. Second, jobs that were already "complete" must also roll back to an idle state if they are needed to reconstruct a lost volume. Thus, the total number of jobs completed dips slightly before resuming its upward course. The machines are restored at approximately 1200 seconds and become available to Hawk.

The workload manager may attempt to provide some insurance against cascading failures, which become quite costly in the case of long pipelines. This can be done simply by checkpointing the state of a container at another node between job executions. If a proxy cache should fail and take a container with it, the pipeline may be resumed from the backup copy. Of course, the creation of the backup copy may come at a significant cost. While many checkpointing systems must ask the user for advice on checkpoint frequency, the manager has a very easy decision: it can monitor the cost of previous backups, record the frequency of job and storage failures, and it knows exactly how much effort was expended to create an existing storage container. Thus, it can easily perform an accurate cost/benefit analysis to determine whether a workload should checkpoint in a given locale.

Choosing an optimal checkpoint interval is an old problem, and Figure 13 demonstrates the continued importance of this decision. Two workloads, consisting of one minute jobs with varying pipeline lengths (l), checkpoint sizes (c), and failure rates were run in our test cluster. The cost/benefit analysis checkpointing policy is compared against always and never checkpointing. When the mean time to failure (MTTF) is low, not checkpointing can lead to slow progress. When the MTTF is high, checkpointing unnecessarily has a significant performance penalty. By applying its knowledge of the workload, the manager can make an appropriate choice.

6 Application Experience

We conclude with a demonstration of Hawk used with a series of real applications that are candidates for high throughput execution on distributed systems. Figure 14 summarizes the I/O behavior of these applications. Each is composed of a number of executable programs that

Policy	Workflow d=10; c=1MB; MTTF=1 hour	d=2; c=1024MB; MTTF=1 day
Always	1629 s	1558 s
Cost/Benefit	1835 s	449 s
Never	2030 s	275 s

Figure 13: **Replication Policies.** *This figure shows the run time of two distinct workflows under three replication policies. Due to its fluctuating estimates of system behavior, cost/benefit is rarely optimal, but always avoids disaster.*

Name	Steps	I/O Traffic (MB) Endpoint	Pipeline	Batch	Total
blast	1	0.12	0.00	355.43	355
cms	2	63.56	12.99	3755.52	3832
hf	3	1.96	4654.34	1.89	4658
amanda	4	5.22	264.31	533.96	803

Figure 14: **Application Summary.** *This figure summarizes I/O needs of four representative scientific applications. The large amount of batch and pipeline traffic I/O for CMS and HF, respectively, are due to multiple reads of shared data of size 120 and 667 MB, respectively.*

form a pipeline, passing data from one stage to the next through the filesystem. In addition, most read data from a fixed set that is shared across many runs. Note that since Hawk migrates executable and library files, their size is included as batch data.

BLAST [3] searches a genomic database for matching proteins and nucleotides. CMS [30] is a simulation of a high energy physics experiment to begin operation in 2006. Hartree-Fock (HF) [14] is a simulation of the non-relativistic interations between atomic nuclei and elections. AMANDA [32] is a simulation of an experimental gamma-ray telescope at the Earth's south pole.

Although many batch scheduling systems do have mechanisms for remote execution, these target applications have not been able to take advantage of them due to their large I/O requirements. However, due to the low ratios of endpoint I/O as shown in Figure 14, most of this I/O could be confined within a migratory file service. Further, these applications present a wide range of program behavior and demonstrate the importance of each of Hawk's three main features of cooperative caching, pipeline containment and fault tolerance. BLAST is a single staged workload that benefits only from cooperative caching. Conversly, HF has heavy pipeline sharing but only very little batch sharing beyond its executable file. AMANDA, with multiple stages and a great deal of batch and pipeline sharing, benefits from all three features.

To measure the throughput for these applications both in remote environments and in Hawk, we used the same configuration as in Section 4 to emulate a local area network of compute nodes and a remote home server. For each application, we compare the throughput of a remote execution system to that of a migratory file service. For all measurements we express throughput as the harmonic mean of the job completion times within that workload.

As one can observe from Figure 15, applications that use Hawk versus the traditional remote I/O solution complete two to three orders of magnitude more jobs per hour.

7 Related Work

Hawk draws on related work from a number of distinct areas. Workflow management has historically been the concern of high-level business management problems involvoing multiple authorities and computer systems in large organizations, such as approval of loans by a bank or customer service actions by a phone company [26]. Hawk's workflow manager 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 [55]. The automatic management of dependencies for both performance and fault tolerance is found in a variety tools [10].

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 [66]; 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 Hawk workflow manager constructs private namespaces for running applications is reminiscent of database views [29]. 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.

The data distribution techniques within the Hawk cooperative cache are quite similar to the distributed hash tables described by Litwin *et al.* [39], and Gribble *et al.* [28]. However, in our implementation, writes are not allowed and thus many of the difficult issues of decentralized update are avoided [51].

There has been much recent work in peer-to-peer storage systems [1, 15, 37, 43, 54, 56]. 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.

There has also been a fair amount of recent work on migrating virtual machines. For example, both Zap [46] and

Name	Throughput (jobs/hour) Remote	MFS
blast	4.67	747.40
cms	33.78	1273.96
hf	40.96	3187.22
amanda	X	X

Figure 15: **Application Throughput.** *This figure shows the end-to-end application throughput for traditional remote I/O and a migratory file service both constrained to 800 KB/s bandwidth to the home server. [Note: The AMANDA results are in progress.]*

VMWare [59] allow for the checkpointing and migration of either processes or entire operating systems. Hawk creates a remote virtual environment for jobs, but at the much higher level of a batch system.

Work in mobile computing also bears similarity to our work on Hawk. For example, Flinn *et al.* discuss the process of data staging on untrusted surrogates for PDAs and other mobile devices [21]. In many ways, such a surrogate is similar to our migratory proxy; the major difference is that the surrogate is primarily concerned about trust, whereas our migratory proxies are primarily concerned about scale and performance. Earlier work on Coda also is applicable [36]. 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 Hawk, a migratory proxies serve a similar role, caching data so as to avoid downtime when the wide-area link to the home node fails.

8 Conclusions

We have introduced Hawk, an instance of a migratory file service. By achieving several orders of magnitude improvement over remote execution, migratory file service offers a fundamentally new way to utilize remote resources. No longer must distributed computing be the sole province of CPU bound jobs with insignificant I/O requirements; using semantic application knowledge and guided by user expectations, a migratory file service recreates the home storage environment of the user on a remote cluster, both in appearance and performance.

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*, Boston, MA, December 2002.

[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] 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.

[4] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *SOSP '01*, Banff, Canada, October 2001.

[5] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, and R. Wang. Serverless Network File Systems. In *SOSP '95*, pages 109–26, Copper Mountain Resort, CO, December 1995.

[6] Avery, P. et al. CMS Virtual Data Requirements. http://kholtman.home.cern.ch/kholtman/tmp/cmsreqsv6.ps, 2001.

[7] B. N. Bershad, S. Savage, E. G. S. Przemyslaw Pardyak, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, December 1995.

[8] D. Bitton and J. Gray. Disk shadowing. In *VLDB 14*, pages 331–338, Los Angeles, CA, August 1988.

[9] W. J. Bolosky, J. S. B. III, R. P. Draves, R. P. Fitzgerald, G. A. Gibson, M. B. Jones, S. P. Levi, N. P. Myhrvold, and R. F. Rashid. The Tiger Video Fileserver. Technical Report 96-09, Microsoft Research, 1996.

[10] 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.

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

[12] S. Chandra, M. Dahlin, B. Richards, R. 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*, October 1997.

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

[14] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, San Diego, California, 1995.

[15] 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, Oct 2001.

[16] 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 *OSDI '94*, Monterey, CA, November 1994.

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

[18] D. A. Edwards and M. S. McKendry. Exploiting Read-Mostly Workloads in The FileNet File System. In *SOSP '89*, pages 58–70, Litchfield Park, AZ, December 1989.

[19] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, December 1995.

[20] 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 *SOSP '95*, pages 201–212, Copper Mountain Resort, CO, December 1995.

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

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

[23] 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.

[24] I. Foster, D. Kohr, R. Krishnaiyer, and J. Mogill. Remote I/O: Fast Access to Distant Storage. In *IOPADS '97*, pages 14–25, San Jose, CA, November 1997.

[25] 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 Tenth IEEE Symposium on High Performance Distributed Computing (HPDC 10)*, San Francisco, California, August 2001.

[26] 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.

[27] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

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

[29] 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.

[30] K. Holtman. CMS data grid system overview and requirements. CMS Note 2001/037, CERN, July 2001.

[31] 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 (TOCS)*, 6(1), February 1988.

[32] P. Hulith. The AMANDA experiment. In *Proceedings of the XVII International Conference on Neutrino Physics and Astrophysics*, Helsinki, Finland, June 1996.

[33] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. Physical File System Backup. In *OSDI '99*, New Orleans, LA, February 1999.

[34] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *SOSP '93*, pages 80–93, Asheville, NC, December 1993.

[35] M. Kim, L. P. Cox, and B. D. Noble. Safety, Visibility, and Performance in a Wide-Area File System. In *FAST '02*, Monterey, CA, January 2002.

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

[37] 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 *ASPLOS IX*, pages 190–201, Cambridge, MA, November 2000.

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

[39] W. Litwin, M.-A. Neimat, and D. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB '94)*, pages 342–353, Santiago, Chile, 1994.

[40] 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.

[41] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, San Jose, CA, June 1988.

[42] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

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

[44] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions of Computer Systems*, 6(1), February 1988.

[45] T. D. Nguyen, R. Vaswani, and J. Zahorjan. Using Runtime Measured Workload Characteristics in Parallel Processor Scheduling. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 155–174. Springer-Verlag, 1996.

[46] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, Dec 2002.

[47] 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 *ASPLOS VIII*, pages 205–216, San Jose, CA, October 1998.

[48] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, IL, June 1988.

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

[50] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *HPDC 7*, Chicago, IL, July 1998.

[51] D. P. Reed. Implementing Atomic Actions on Decentralized Data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

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

[53] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[54] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *SOSP '01*, Banff, Canada, October 2001.

[55] 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.

[56] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *OSDI '02*, Boston, MA, December 2002.

[57] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[58] 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.

[59] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, Dec 2002.

[60] M. Satyanarayanan. A Survey of Distributed File Systems. Technical Report CS-89-116, CMU, Pittsburgh, Pennsylvania, 1989.

[61] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 1996.

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

[63] 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.

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

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

[66] A. Vahdat and T. E. Anderson. Transparent Result Caching. In *USENIX '98*, New Orleans, LA, June 1998.

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