Operating System Support for Space Allocation in Grid Storage Systems

Douglas Thain
University of Notre Dame

Abstract— **Shared temporary storage space is often the constraining resource for clusters that serve as execution nodes in wide-area distributed systems. At least one large national-scale computing grid has reported a failure rate of as high as thirty percent of submitted jobs, often due to accidentally filled shared storage spaces. Previous systems have attacked this problem by adding space allocation to the distributed system interface. However, these allocations are not enforced at the filesystem level, and thus unexpected or unaccounted uses of storage may cause the system to fail. By adding an inexpensive allocation mechanism to the operating system, we may improve the robustness of such systems at minimal cost. In this paper, we describe an abstract model of space allocation in the file system and explore three implementations of the model: a user-level library, a recursive loopback filesystem, and a modified kernel filesystem. We evaluate the performance and completeness of these implementations and demonstrate that kernel support is essential to keeping the overhead low. Finally, we demonstrate empirically that a cluster under heavy filesystem load can be made more robust by adding allocations to the filesystem.**

I. Introduction

Shared temporary storage space is often the constraining resource for clusters that serve as execution sites in wide-area distributed systems. For many reasons, users find themselves inadvertently sharing limited storage space. This may be for administrative convenience: all home directories might be stored on a single file server. Or, it may be due to security constraints: a firewalled cluster might require users to stage input and output data on a single node before moving it elsewhere. Or, it may be due to pure chance: two users may happen to store a large dataset on the same worker node. Without the ability to allocate storage space, users with long running or large data sets encounter trouble. The careful user may check for available space and then start a job or data transfer only to discover that someone else has gobbled it up before the task can complete. Such users must retreat to private, overprovisioned resources to accomplish their work.

An example of this problem has been observed in the context of Grid3, the predecessor of the Open Science Grid. In 2003, Grid3 comprised several clusters located at tens of research institutions in the United States, serving the needs of hundreds of researchers on thousands of processors. However, there was observed a remarkable failure rate: thirty percent of jobs submitted over a period of several months failed. [1] Ninety percent of these failures were due to a full shared disk. One unchecked log file, one crashed job, or one careless user could easily fill a shared disk, causing cascading failures on all other users of that disk. Of course, this problem is not unique to Grid3; this is only one well-documented example. Users of grid systems know that such problems are common, aggravating, and difficult to diagnose and solve.

If it were possible to allocate disk space in the same manner as one may allocate other computing resources, many of these problems would be mitigated. Jobs in a batch system could be allocated space before they begin so that started jobs would have a much higher probability of finishing. Jobs denied allocations could be delayed, placed elsewhere, or returned with a clear error message. System services would not be affected by misbehaving jobs, or vice versa.

Existing storage management systems (e.g. SRM [2], SRB [3], IBP [4], NeST [5]) offer interfaces that allow users to request allocations before consuming space. However, these systems have no mechanism in the operating system to enforce allocations against local competitors short of consuming the space immediately. This is an acceptable situation if the manager is aware of all users of storage, and all users can be trusted to operate correctly. However, these two assumptions do not hold in a grid computing system. Computational jobs cannot be trusted: they may be submitted by malicious users, they may crash and generate large core dumps, or they may be given accidental arguments that generate large outputs. Not all storage uses are known: local services generate log files, incoming emails consume space, and local users may perform tasks irrelevant to the wide area system. Both running jobs and local services may compete for space.

Of course, administrators have some tools for managing disk space: they may partition disks and establish user and group quotas. However, neither of these mechanisms is well suited for protecting jobs running in a distributed system. Partitions are limited in number and are expensive to create and manage. User and group quotas are inexpensive to apply, but require that ownership of files and processes align with allocation needs, but this is rarely the case in grids: a single user might wish to have many small allocations, while many users might wish to share a large allocation. Both mechanisms require the storage manager to run with special privileges.

Therefore, we propose that space allocation should be provided *in the lowest levels of the filesystem* on shared storage devices. In particular, the allocation mechanism must allow ordinary users to make fine-grained allocations of space in a manner tightly coupled with the namespace. Each of these properties is important: the mechanism must be accessible to *ordinary users*, so that robust services can be constructed without requiring superuser privileges. It must be *fine-grained*,

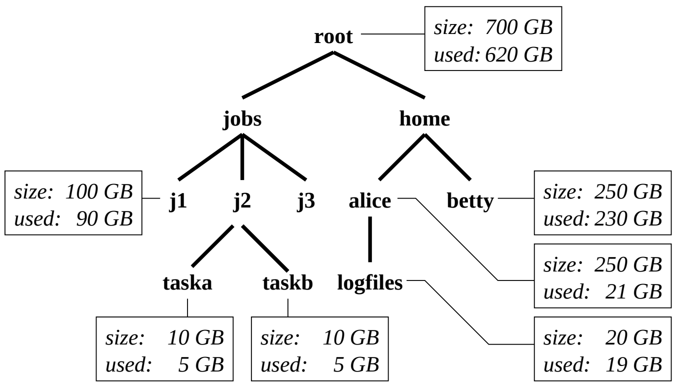


Fig. 1. An Allocable Filesystem

allowing for a separate allocation to be made for many jobs initiated by the same user. It must be an *allocation* that offers both a guarantee of available space, as well as an enforcement from consuming too much space. It must be *tightly coupled* with the namespace so it is easy to use.

In this paper, we describe an abstract model of allocation in the filesystem, using hierarchical accounting and the addition of three system calls. We describe how this basic form of allocation may be used as a building block for more complex distributed systems. We explore three implementations of this model: a user-level library, a loopback filesystem, and a modified kernel filesystem. Each of these implementations has various strengths and weaknesses. The user-level library may be applied on any system without modification, but requires the explicit participation of all parties. The loopback filesystem can be applied transparently, but requires root privileges, has limited applicability, and is expensive to create and delete. The modified filesystem requires kernel changes to deploy, but is transparently applied and has very low overhead. Finally, we deploy an allocable filesystem on a cluster and demonstrate that allocations improve robustness by maintaining consistent throughput under high offered load.

II. A Model of Allocation

We begin by defining a model of space allocation that can be implemented on a passive storage device.

The model must have the following properties:

Unprivileged use. In large scale systems, it is very rare for a user to have administrative privileges on all, or even a few machines. Operating as an ordinary user is the norm. In addition, the security-conscious administrator will wish to run system services with the minimum privileges needed. Thus, the allocation mechanism must be accessible to ordinary users, within the constraints of access controls on the filesystem.

Fine granularity. The value of data is not apparent from its size. Many CPU-days of computation may be necessary to produce a few KB of valuable data, whereas it is very easy to fill a disk with many gigabytes of garbage. A single user may have many independent jobs, each requiring their own, possibly small, allocation. Thus, the mechanism should have a reasonably small overhead in time and space so that it may be used to protect small amounts of data.

Prevention and Guarantee. An allocation must serve two distinct purposes. It must prevent a job from overrunning the allocation, so that it does not cause other jobs to fail. But, it must also guarantee that a job will have access to the entire space of its allocation, otherwise the job itself will fail. Both properties are necessary for a robust system.

Namespace coupling. Many users may access an allocable filesystem through conventional Unix interfaces. They may not necessarily be written to manipulate or even view the state of allocations. Thus, where possible, the nature of allocations must be coupled to the visible namespace of the filesystem, so that file operations such as create, rename, and delete have a reasonable and expected effect upon the allocation state.

A. Allocation Primitives

An allocable filesystem is a hierarchy of directories, like a conventional filesystem. Each directory may be an ordinary directory, or it may also be an *allocation*. An allocation accounts for the storage consumed by all files, directories, and allocations contained within it. An allocation contains an *allocation state* which has two fields: *size* records the fixed size of the allocation and *used* reflects the amount consumed by data and sub-allocations. Ordinary files and directories consume space in the nearest ancestor allocation. That is, if a user writes to a file, the *used* field of the nearest ancestor allocation increases by the size of the data written. *used* may grow until it reaches *size*. If this happens, any further writes fail immediately with an appropriate error code. Conversely, deletions of data and allocations decrease the *used* field.

A newly created filesystem has a root allocation such that *size* is the total disk size and *used* is zero. In order to create a new allocation, a user may issue the *mkalloc(path,size)* system call. This causes two things to happen atomically: (1) A new directory is created in *path* with allocation state *(size,0)*. (2) The *used* field of the nearest ancestor allocation state of *path* is decreased by *size*. Of course, such an operation may only be performed if the following conditions hold: (A) The user must have permission to call *mkdir(path)*. (B) *(size-used)* of the nearest ancestor must be greater than *size*.

An allocation may be deleted with the ordinary *rmdir(path)* on a directory with an allocation state. (The filesystem may also require the user to remove the contents of the directory before the directory itself.) This causes two things to happen atomically: (1) The directory and its contents are deleted. (2) The *size* field of the deleted directory is subtracted from the *used* field of the most immediate ancestor allocation.

An allocation may be adjusted by a call to *mkalloc* with an existing path and a new allocation size. This causes two things to happen atomically: (1) The *size* of the existing allocation is adjusted. (2) The new *size* is added to the *used* field of the immediate ancestor. This will only succeed if the requested *size* is larger than *used* and if there is sufficient space.

An allocation may be examined by a call to *lsalloc(path)* which returns a *(apath,size,used)* tuple indicating the path of the containing allocation, the total size, and the space used.

Note that the creation, deletion, and modification operations are both *atomic* and *idempotent*. An operation will either succeed or fail entirely, and operations may be repeated many times with the same effect as once. Thus, a transaction interface is not needed. If an allocating entity fails to receive a response from an operation, it may simply issue *lsalloc* to learn the current state, and retry the operation if needed.

Finally, note that allocation is orthogonal to a traditional Unix quota system. Allocations limit space consumed per directory, while quota limits space consumed by a user across an entire filesystem. Both mechanisms may be simultaneously active on a single system.

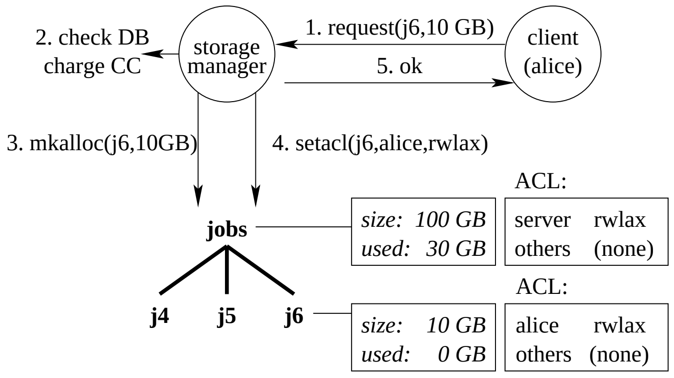


Fig. 2. External Storage Manager

B. Common Applications

Allocations are useful to many stakeholders in the system: administrators, interactive users, batch users, and system services. Figure 1 suggests several different uses for allocations.

Allocation is useful to *administrators*. In Figure 1, the system administrator has used *mkalloc* to set aside a fixed amount of space (250GB) for the home directories of the users alice and betty. Of course, administrators already may perform simple allocations through partitions, and quotas. However, the administrator gains several advantages when using an allocable filesystem. (1) Unlike partitioning, allocation does not require drastic actions such as remounting, rebooting, or reformatting. (2) Unlike quotas, allocation is not coupled to user IDs: allocations may be given to jobs, tasks, projects, or other entities. (3) Allocation may be delegated to other staff without root privileges, merely by giving the appropriate write permissions on the filesystem.

Allocation is useful to *interactive users* because it allows them some protection against misbehaved or resource hungry applications. For example, suppose that Alice is dispatching a large number of jobs to a computational grid. She runs some local scheduling software that must communicate with remote nodes, dispatch jobs, log actions, and collect output files. There are many ways such a tool can accidentally consume all of her space. By running it within an allocation (*logfiles*), she can prevent it from overrunning her other activities.

Allocation is useful to *batch users* because it allows the user some protection from (and avoidance of) the vagaries of large distributed systems. Figure 1 suggests that the filesystem is used to store the files needed by three batch jobs in directories j1, j2, and j3. If the submitter of the job is able to state the job's storage needs in advance, an allocation can be made for the job, giving it a greater assurance of executing to success. If the allocation fails, the job would be likely to fail anyhow, so an intelligent scheduler may attempt to allocate elsewhere. A job with unknown storage needs may still be executed, without an allocation, but it receives no guarantees.

Allocation is useful to *system services* because it allows for the protection of local resources from visiting users. From the perspective of a service provider, visiting jobs are dangerous because they may exhaust resources already assigned to local users. Betty would be justifiably upset if a visiting job consumed space intended for her home directory. By employing allocations, local systems can be made robust against the attacks of distributed systems.

C. No Allocation Policy

This model deliberately has no intrinsic policy governing how much a user may allocate or how long they may hold the allocation, beyond the two basic constraints: (A) The user must have permission to perform *mkdir*. (B) The new allocation must fit in the ancestor. The reason for this is simple: we cannot imagine a policy language that would satisfy all users, nor is there any clear way to store it in the filesystem proper. Moreover, many desired policies could not be implemented within one filesystem. For example, a policy that limits a user to consuming a total of 1 TB of allocations across a cluster of ten devices cannot be implemented within one device.

If a policy beyond the two simple requirements above is desired, an external mechanism is required. A filesystem to be allocated to external users is given an ACL granting write permissions only to to the owner of a *storage manager* (e.g. SRM [2], SRB [3], IBP [4], NeST [5].) A client requiring space contacts the storage manager, which then consults databases, queries a human administrator, charges a credit card, or does whatever is necessary to authorize the request. If permitted, the policy manager performs *mkalloc* on the filesystem. The directory will initially only allow access to the policy server, so it must then modify the permissions to give the calling user appropriate access. Once this directory is allocated, the calling user is free to employ it in any way, including subdividing it into further allocations. If the allocation has a time limit, then the storage manager may implement the necessary action when the time has expired. This action might be as simple as sending an email, or as drastic as deleting the allocation outright.

D. Common Questions Answered

Thoughtful readers readers have posed several common questions about allocation in the filesystem.

Q: Do allocations allow a malicious user to halt the system by allocating everything? A: No. A malicious user that at-

	need root to install?	**need root to alloc?**	**install in place?**	**limit space?**	**guarantee space?**	**to create allocation**	**to delete allocation**	**recover alloc**
library	no	no	yes	yes	no	create file	rm file	traverse
loopback	yes	yes	no	yes	yes	dd + mkfs	rm large file	fsck
kernel	yes	no	yes	yes	yes	update inode	update inode	traverse

Fig. 3. Comparison of Three Implementations of Allocation

tempts to allocate all remaining space may do so, but this will not crash other services. When a system is properly configured, key system services are given an appropriate space allocation for their needs. A malicious user might be able to consume all of the *remaining* space, but existing allocations will not be disrupted. Note that this is an improvement on conventional systems that allow any user to exhaust all available space with *cat /dev/zero > /tmp/bigfile*. In addition, consider that it may be *appropriate* for a benevolent user to allocate all available space: If a user has a job that requires 500 GB and a single 500 GB disk is available, then the allocation *should* prevent others from using the space so that the job may complete.

Q: Don't allocations create a garbage collection problem? A: No. Ordinary filesystems already have a garbage collection problem, and allocations neither aggravate nor eliminate the problem. Any system that makes use of files must perform some sort of application-specific garbage collection: batch systems must delete scratch directories when jobs complete; mail systems must delete or bounce undeliverable mail; administrators must delete home directories when users depart. The use of allocations for these tasks does not eliminate the garbage collection problem; it simply helps to ensure that storage resources are not overcommitted. Note also that many storage managers implement garbage collection: for example, IBP requires every allocation to have a finite lifetime.

Q: How do allocations interact with jobs that have unknown or variable space needs? A: Allocations neither help nor hurt jobs with unknown needs. A user that does not wish to make use of allocations may simply make use of a filesystem in the ordinary fashion. Just-in-time and advance allocation of space may be supported in the same system. However, both on-demand and advance allocation may be served simultaneously. Consider again the filesystem in Figure 1 above. One job has requested and received an allocation in *j1*. *j3* has no allocation, perhaps because it has unknown needs. *j2* has unknown overall needs, but knows that it needs 10GB for each of two subtasks. When allocation used selectively in this manner, allocation protects the jobs with known needs, but does not harm jobs with unknown needs, which are already exposed to some risk.

III. Three Implementations Compared

We have implemented the abstract model of allocation in three ways: a user level library, loopback filesystems, and a kernel-level driver. Figure 3 summarizes the major differences.

A. User-Level Library

The first implementation is a user level library that provides an I/O interface similar to that of Unix system calls. For example, entry points are *alloc_open()*, *alloc_read()*, *alloc_mkdir()*, and so forth. Programs may be modified to take advantage of this library, or the library may be transparently inserted using interpositioning techniques.

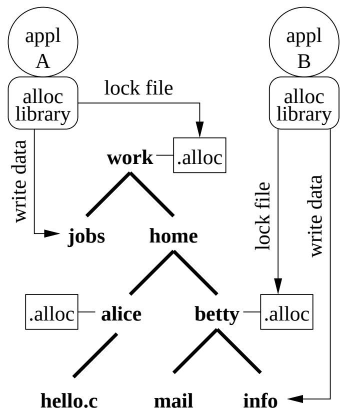
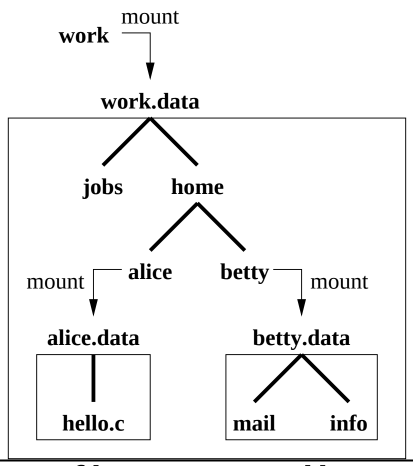


Fig. 4. Library Implementation

The library maintains allocation state by adding an extra file *.__alloc* to each directory that is an allocation. Each file records the *size* and *used* fields described above. Read-only I/O operations are executed without modification. However, write I/O operations must check and update the state of the allocation file on disk. Because multiple independent processes may be accessing the same filesystem at once, advisory locks are used to ensure that the allocation state is not corrupted.

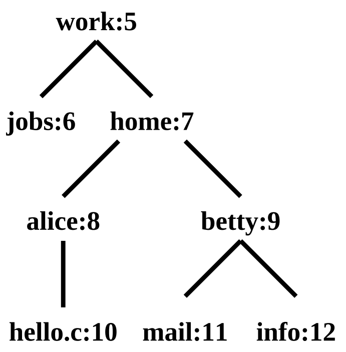
The primary obstacle to good performance is the locking mechanism. In the worst case, a single write to a file results in a lock, open, read, write, close, and unlock on the allocation file. This would cause an order of magnitude increase in the latency of write operations. To address this, the library caches locks on allocation files, and then only writes and unlocks them when the client has been idle for some time (default is two seconds.) Thus, under sustained operations, the overhead of write operations is only an *fstat* necessary to obtain the current size of the file, plus several memory operations to update the cached state. The drawback to caching allocation locks is that other clients waiting to access files in the same allocation may experience delays. Thus, this mechanism is best suited to applications where there is a maximum of one process operating on an allocation.

Deadlock is always a concern with multiple processes. Most operations only require access to a single state file at one time. To avoid deadlock, a process that has cached several locks performs a non-blocking lock when requesting a new lock. If the lock fails (another process holds it) then all cached



filesystem mount table:		
mount point	**block device**	**backing file**
/	/dev/hda1	-
/work	/dev/loop0	/work.data
/work/alice	/dev/loop2	/work/alice.data
/work/betty	/dev/loop4	/work/betty.data

Fig. 5. Loopback Implementation



filesystem inode table:

inode	size	inuse	parent
5	1 GB	500 MB	0
6	-	-	5
7	-	-	5
8	200 MB	10 MB	5
9	300 MB	20 MB	5
10	-	-	8
11	-	-	9
12	-	-	9

Fig. 6. Kernel Implementation

locks are released before attempting another lock. This breaks the hold-and-wait condition. In the few cases where multiple simultaneous locks are required (e.g. a rename() between allocations), locks are obtained in order from lowest to highest inode number. This breaks the circular wait condition.

Because locks are cached in memory, we must accept the possibility that the allocation state may be out of sync with stored files after a crash of a process or the machine. Much as with a traditional filesystem, the library marks the allocation root dirty on startup and marks it clean only on an orderly shutdown. If the library encounters a dirty root allocation, then it must rebuild the *inuse* fields of the allocation states by recursively traversing the filesystem and measuring file sizes.

The user-level library has several advantages, particularly in deployment. Because it stores allocation state in hidden files, it can be applied without any special privilege and can even be deployed over an existing filesystem by simply adding a few *.__alloc* files. However, respect for user-level allocations is voluntary. An allocation-enabled program will not overflow its stated allocation, but it has no guarantee of success when competing against non-enabled programs.

B. Recursive Loopbacks

Allocations can also be implemented by creating a loopback filesystem for each allocation. A loopback filesystem is an ordinary filesystem in every respect, except that it uses an ordinary file – rather than a block device – as a backing store. Virtual disk images created for virtual machines are nearly identical in concept and implementation.

On Linux, a loopback is created by generating a large file using *dd*, connecting a loopback device */dev/loopN* to that file using the *losetup* tool, formatting the filesystem with *mkfs* and then mounting the filesystem into directory hierarchy. Similar mechanisms are present on other Unix-like operating systems. Loopback filesystems may be generated recursively in order to implement hiearchical allocations. That is, a backing store may be created within an existing loopback filesystem, and then used to store and mount a new loopback.

Loopback filesystems are a heavyweight but robust method for generating allocations. Typically, a small number (10s) of loopback devices are configured into the kernel, and one must be *root* to configure and mount them. Once configured, ordinary users may take advantage of loopback filesystems using the ordinary protection mechanisms. Because of the limited number of devices, a system that requires a large number of allocations in the filesystem would not be able to use all simultaneously. Loopbacks must be created from scratch, so it is not possible to retrofit allocations to an existing filesystem. However, loopbacks do provide both limits to space consumption and a guarantee of available space because all blocks are allocated when the filesystem is created. Creating and deleting loopback filesystems can be very expensive: because each block must be allocated or returned to the filesystem, these operations are linear with respect to the size of the allocation. To recover from a crash, *fsck* must be run recursively on each allocation.

C. Kernel Filesystem Driver

The underlying problem with loopback filesystems is that the kernel provides the user no method for allocating disk blocks from the filesystem short of actually writing to each block directly. In order to have fast, robust allocations, we must have in kernel support for allocating and releasing disk blocks without actually touching them. To demonstrate this, we have modified a standard Linux *ext2* filesystem to support allocations. We call the resulting filesystem *allocfs*. This filesystem operates in a manner similar to the user-level library, except that the allocation state is stored in inode structures. The *ext2* filesystem has several unused fields in the inode, which we use to store the *size* and *inuse* fields described in the abstract model. In addition, a third field *parent* is used to record the inode of the parent allocation. The *mkalloc* and *lsalloc* system calls simply consult and update fields in the inode as described in the abstract model.

To enforce allocations, the block allocation routines are adjusted to consult the appropriate inodes for available space before examining the free block bitmap. The extra cost for enforcing allocations involves at most fetching and replacing the cached parent inode, plus several memory operations. However, because the parent inodes of files and directories in use are generally already cached for other reasons (e.g. due to `namei`), this cost is essentially free.

It is important to note that this approach blurs the conventional distinction between inodes and directory structure. As a result, some operations must be rejected because they introduce inconsistencies into the allocation state. For example, we do not wish to have multiple directory entries from different allocations linked to the same inode: which allocation would be charged for the space? As a result, hard links that cross allocations are not permitted. In addition, directory trees cannot be renamed from one allocation to another, as this would require the kernel to recurse over the subtree to determine the total space used in order to update the necessary allocation states. However, individual files and allocations can be renamed, as the total size is readily available.

Three user-level tools are provided to manipulate allocations. *mkalloc* <*path*> <*size*> creates or adjusts the allocation state of a directory, within filesystem permissions. *lsalloc* <*path*> returns the containing allocation path and state of a given file or directory. *fixalloc* <*path*> is used by the superuser to check and recover the allocation state of the filesystem after a crash. In order to implement *fixalloc*, a third system call is necessary: *setalloc(fd,size,used,parent)* sets the absolute values of the allocation states of a directory. *setalloc* should only be used by the superuser to repair a filesystem.

The modified kernel driver provides the best performance and semantics of the three implementations, but the primary drawback is deployment: the filesystem must be configured and installed by *root*. However, the *allocfs* filesystem is binary compatible with *ext2*: it may be used to mount an existing filesystem and add allocation state to it. Allocations both prevent overruns and guarantee available space. Creation and deletion of allocations only involves updating inode state. Like the user-level library, allocation state must be recovered by a recursive directory traversal.

IV. Performance Evaluation

We compare the performance of the three implementations by examining I/O latency, I/O bandwidth, and allocation operations on the same disk. All measurements are taken on a 40GB Seagate ST340015A ATA disk with 2MB cache, Intel 82801EB ATA controller in PIO mode, 2.8GHz Pentium 4 CPU with 1GB of memory running Linux 2.4.21. For each test, a fresh EXT2 filesystem is created, and the buffer cache is cleared by reading 1GB from another disk. Loopback allocations further create a large file on the fresh filesystem, and then format it as an EXT2 filesystem.

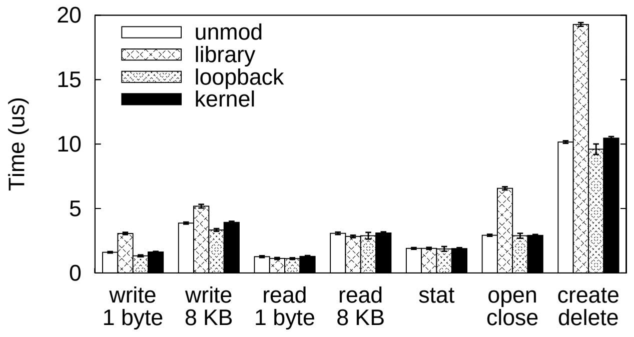


Fig. 7. Latency of I/O Operations

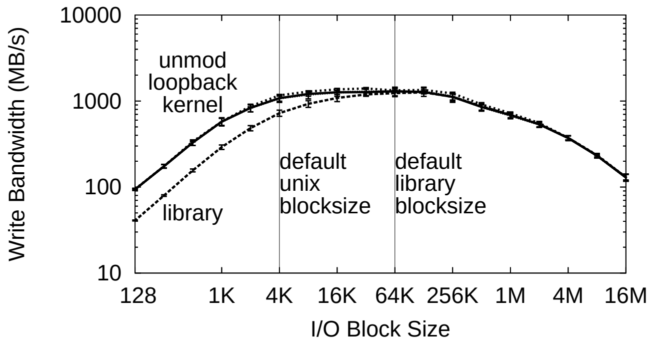


Fig. 8. Write Bandwidth

Figure 7 compares the latency of basic I/O operations in each of the allocation types. Each operation is attempted in 100 timed loops of 10,000 system calls, average and standard deviation are shown. Several things should be noted about the results. The user-level library imposes some overhead on write operations, due to the required extra *fstat* described above. Read operations are unaffected. Surprisingly, directory operations in loopback are *faster* than the unmodified filesystem! This is because synchronous metadata updates in the loopback are eventually reduced into asynchronous data updates in the backing file. This is arguably incorrect behavior: metadata updates are usually required to be synchronous so as to maintain consistency of the directory structure.

The performance of creating allocations is as follows:

loopback	1 sec per 25 MB of allocation
library	227 μsec regardless of size
kernel	32 μsec regardless of size

Creating a loopback allocation requires writing to every single block, and will take several minutes for allocations measured in GB. In the library and kernel, creating an allocation is comparable to creating a directory.

Figure 8 compares the disk bandwidth available to applications while writing a 100MB file using various block sizes. (In this context, the block size is the *length* parameter to a *write*

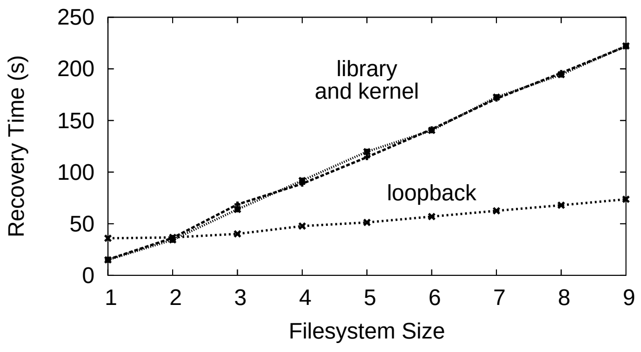


Fig. 9. Allocation Recovery Time

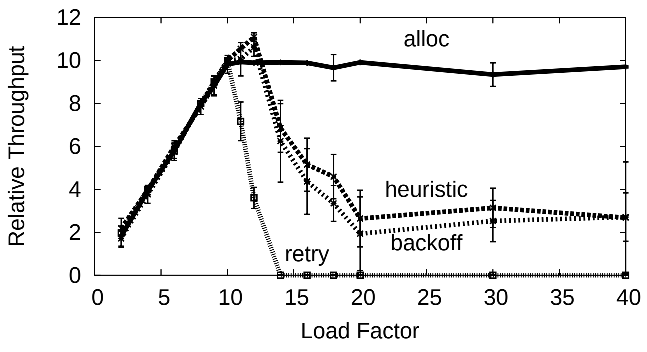


Fig. 10. Robustness Under Load

system call at user level.) Each measurement is taken ten times, average and standard deviation are shown. For sufficiently large block sizes, all methods are able to obtain the same bandwidth from the file system. The user-level library has a fixed overhead (*fstat* again) for each write operation and thus cannot achieve the full bandwidth for small block sizes. However, because the library also interposes on the *fstat* call, it may overcome this by hinting a larger block size to the standard I/O library.

Figure 9 shows the time to recover the allocation state of dirty filesystems of various sizes. Filesystems are created by copying a clean Linux source tree of 884 directories, 14869 files, and 201 MB of data blocks. Each unit on the X axis indicates one additional copy of the source tree. The user-level library and the kernel filesystem recover by traversing the directory structure, while the loopback filesystem recovers by issuing a *fsck in addition* to the *fsck* required on the host filesystem. Note that the loopback has a higher fixed cost but lower marginal cost as the data in the filesystem increased. This is because the first phase of *fsck* must scan all inodes – whether in use or not – to build a free block bitmap. However, later phases that traverse the directory tree are faster than the equivalent operation from user space. This is due to the movement of data in and out of the kernel: The directory traversals require several user-kernel switches for each directory and file to be examined, while *fsck* can request one large block of data and use it to examine several structures at once.

Finally, we demonstrate that allocation improves the robustness of a cluster where disk space is the resource constraint. We establish a cluster emulating an execution site on a computational grid. Multiple CPUs share a file system on the head node. To execute a job, input data must be copied to the data must be staged to the shared disk, the job must be run, and then its output must be copied elsewhere. This model is used by clusters connected to computational grids such as the Open Science Grid, the NSF TeraGrid, and the European Data Grid. For example, one may use GRAM [6] to execute jobs on a remote cluster while relying on GridFTP [7] to stage input and output data for the job.

To explore a controlled environment, we construct synthetic jobs that steadily produce 100MB over one minute. As each completes, the output file is transferred elsewhere and then deleted. The shared disk has 1000MB of space available, so the maximum safe load is ten jobs running simultaneously. We assume that, like most scientific applications, output errors are not carefully checked at runtime, so a failure to write will only be detected after the job completes.

Without an allocation service in the filesystem, the caller has several possible techniques for dealing with a lack of storage. The caller may simply blindly start jobs and immediately **retry** if they should fail. A more responsible caller will perform **backoff** and delay an exponentially-increasing amount of time (plus a random factor) between failures. This technique allows temporary resource overloads to "even out" over time. A more responsible approach is to use a **heuristic**: the caller may check to see if sufficient disk space is available before dispatching a job. A heuristic is not guarantee: if the job should still fail, then backoff is used. Finally, we may compare these techniques to using **allocation**: the caller allocates space before running the job and releases the space when done. If the allocation fails, the caller waits one second and tries again.

Figure 10 shows the throughput of a cluster under load using the four management techniques. The X axis shows the offered load in number of active CPUs on the cluster. The Y axis shows the cluster throughput, normalized to a load of one. When the load is less than ten, storage space is overprovisioned, and the speedup is linear. As the load increases over ten, the performance of the techniques diverges. Simple **retry** quickly crashes to zero throughput: no jobs are able to run to completion once storage is overcommitted. Both **backoff** and **heuristic** are able to maintain some throughput, but get worse as load increases. As expected, **allocation** maintains full throughput even under heavy load.

Note that **backoff** and **heuristic** are able to exceed the throughput of **allocation** for critical values slightly above the maximum safe load of the system. Because each job consumes storage gradually, it is possible to slightly overcommit storage while avoiding failures. Thus, allocation does *not* provide maximum possible performance: it provides predictable performance across a range of loads.

V. Related Work

There exist a variety of mechanisms that provide allocation-like services in file and distributed systems. For example, the basic allocation unit is a *volume* in AFS [8], and a *partition* in OSD [9]. Many commercial network storage devices allow for internal re-partitioning. However, these mechanisms are relatively expensive, only available to the superuser, and cannot be subdivided. Virtual machines [10], [11], [12], [13] allow one to allocate space in the form of a private virtual disk for contained processes. This is very similar to the loopback filesystem and has similar performance properties. ZFS [14] allows for the creation of new filesystems as allocations within the directory hierarchy, but these allocations are neither hierarchical nor manageable by ordinary users. Resource containers [15] are closer in spirit to our work: they allow related processes to share a set of resources such as memory and CPU time. However, they do not address the persistent nature of storage.

Our notion of allocation is inspired partially by the Internet Backplane Protocol (IBP) [4], which proposes a malloc-like interface to raw storage. Using IBP, consumers of distributed storage may request storage extents with renewable time limits and a capability-based protection model. This abstraction is simple enough that it can be implemented in relatively simple devices, but is sufficiently powerful to construct higher level storage abstractions such as the ExNode [16]. However, large scale grid computing must engage existing applications that make heavy use of existing interfaces such as the Unix filesystem. It is not practical (in the short term) to design new interfaces entirely from scratch. Thus, this work aims to migrate the concept of storage allocation in IBP into the familiar interface of the filesystem, where it can be used by unmodified applications. It may be possible to implement filesystem applications on top of IBP or vice versa.

The role of a storage manager that we have outlined above may be fulfilled by a variety of grid computing tools. For example, the Storage Resource Manager (SRM) [2] defines an interface that allows users to request storage with certain size and temporal properties. The SRM is responsible for negotiating with the underlying storage and then directing the user to the underlying device, which is responsible for enforcing the allocation. The NeST [5] storage appliance has a form of allocation known as *lots*, which exist in a namespace distinct from the filesystem. A similar role is played by components of Freeloader [17] and the Storage Resource Broker [3].

VI. Conclusion

Ten years ago, Lepreau et al. argued that the facilities provided by the local operating system are critical to creating robust distributed systems[18]. This argument is still relevant today: large scale distributed systems are present and yet maddeningly unreliable [19], partially because the necessary facilities are not found in the operating system. This work is a focused attempt to address one limitation of local systems: the ability to allocate storage within the context of a filesystem. We have described a model of allocation useful to both end users and system designers. Although varying implementations are possible, we have shown that in-kernel support for allocations provides the best performance along with semantics not available from the user-level.

Acknowledgement

Paul Madrid assisted with the user level library.

References

[1] R. Gardner and et al., “The Grid2003 production grid: Principles and practice,” in *IEEE Symposium on High Performance Distributed Computing*, 2004.

[2] A. Shoshani, A. Sim, and J. Gu, “Storage resource managers: Middleware components for grid storage,” in *Nineteenth IEEE Symposium on Mass Storage Systems*, 2002.

[3] C. Baru, R. Moore, A. Rajasekar, and M. Wan, “The SDSC storage resource broker,” in *Proceedings of CASCON*, Toronto, Canada, 1998. [Online]. Available: citeseer.nj.nec.com/baru98sdsc.html

[4] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski, “The Internet Backplane Protocol: Storage in the network,” in *Network Storage Symposium*, 1999.

[5] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny, “Flexibility, manageability, and performance in a grid storage appliance,” in *Eleventh IEEE Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.

[6] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, “Resource management architecture for metacomputing systems,” in *IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing*, 1998, pp. 62–82.

[7] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, and S. Tuecke, “Protocols and services for distributed data-intensive science,” in *Proceedings of Advanced Computing and Analysis Techniques in Physics Research*, 2000, pp. 161–163.

[8] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, “Scale and performance in a distributed file system,” *ACM Trans. on Comp. Sys.*, vol. 6, no. 1, pp. 51–81, February 1988.

[9] M. Mesnier, G. Ganger, and E. Riedel, “Object based storage,” *IEEE Communications*, vol. 41, no. 8, August 2003.

[10] A. Whitaker, M. Shaw, and S. D. Gribble, “Scale and performance in the Denali isolation kernel,” in *Operating System Design and Implementation*, Boston, MA, December 2002.

[11] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, “Xen and the art of virtualization,” in *Symposium on Operating Systems Principles*, 2003.

[12] J. Dike, “A user-mode port of the Linux kernel,” in *USENIX Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[13] A. Bavier, S. Karlin, S. Muir, L. Peterson, T. Spalink, M. Wawrzoniak, M. Bowman, B. Chun, T. Roscoe, and D. Culler, “Operating system support for planetary scale network services,” in *Networked Systems Design and Implementation*, 2004.

[14] *Solaris ZFS Administration Guide*. Santa Clara, CA: Sun Microsystems, May 1996.

[15] G. Banga, P. Druschel, and J. C. Mogul, “Resource containers: A new facility for resource management in server systems,” in *Operating Systems Design and Implementation*, 1999.

[16] M. Beck, T. Moore, and J. Plank, “An end-to-end approach to globally scalable network storage,” in *ACM SIGCOMM*, Pittsburgh, Pennsylvania, August 2002.

[17] S. Vazhkudai, X. Ma, V. Freeh, J. Strickland, N. Tammineedi, and S. Scott, “Freeloader:scavenging desktop storage resources for scientific data,” in *International Conference for High Performance Computing and Communications (Supercomputing)*, Seattle, Washington, November 2005.

[18] J. Lepreau, B. Ford, and M. Hibler, “The persistent relevance of the local operating system to global applications,” in *SIGOPS European Workshop*, 1996.

[19] J. Schopf, “State of grid users: 25 conversations with UK eScience groups,” Argonne National Laboratory Tech Report ANL/MCS-TM/278, 2003.