RC24124 (W0611-190) November 28, 2006
Computer Science

IBM Research Report

Evaluating Stackable Storage as the Building Block for Systems Management

Eric Van Hensbergen
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758

Gorka Muzquiz
Ray Juan Carlos University

Fabio Oliveira
Rutgers University

Evaluating Stackable Storage as the Building Block for Systems Management

Abstract

Centralized systems management approaches have been recently investigated in the academia and implemented in the industry aiming to facilitate the deployment of disk images to groups of machines. The old approach of installing and configuring each machine individually is most of the time impractical and error-prone. However, no study to date has sought to investigate different design alternatives for storage systems supporting centralized disk image management, or evaluate their performance and scalability. In this paper, we first propose the notion of *stackable storage* as a mechanism to organize disk images on centralized storage servers, and show how this mechanism would help the system administrator to perform typical management tasks. We next discuss different design options by varying three key components of the *stackable storage* design space. Finally, we prototype the design options and evaluate their performance behavior and scalability.

1 Introduction

The computing infrastructure on which small and medium-sized firms rely is becoming increasingly complex. Many such companies depend on software components distributed over a number of servers running internal application suites or providing on-line Internet services. These distributed components include database servers, email servers, identity management servers, application servers, Web servers, load-balancers, and many others. This diversity of mutually dependent software complicates the fundamental tasks of systems management: initial installation, software upgrade, new hardware deployment, and coherent cluster configuration.

A descentralized systems management approach is still adopted by some organizations, where each machine is individually installed from CD-ROM or DVD media and manually configured. If the administrator needs to deploy new hardware to expand the system capacity, or to reprovision a cluster to balance the workload, the tedious process of installation and configuration has to be repeated, making sure that all necessary software components are installed on each node — an error prone task at best —, not to mention the need to keep all machines up-to-date with the most recent patches. That humans invariably make mistakes worsens the problem. Several studies have shown that human mistakes committed during system administration were the predominant cause of service disruptions and outages [12, 22, 21, 20].

It comes as no surprise that centralized systems management mitigates the aforementioned problems and significantly reduces the management costs. In light of this observation, approaches to centralized management have been investigated in the academia [6, 32] and implemented in the industry [3, 30], aiming at facilitating disk image manipulation and deployment by means of sharing. However, thus far no study has investigated different design alternatives of centralized storage servers for the purpose of disk image management, evaluated them, or characterized their performance behavior and scalability.

In this paper, we address this issue in three steps. We first propose a mechanism for centralized disk image management which we call *stackable storage*. The name *stackable storage* stems from the idea of composing different views of the storage system by stacking layers of disk image. This mechanism makes it easy to segregate user or machine-specific data from OS and application images, a characteristic on which most existing approaches to centralized management rely. Next, we identify different policies for the stackable storage mechanism by varying the granularity at which disk image layers can be stacked — block level or file system level —, the target of the stacking code — storage server or clients —, and the type of data transport protocol used by the clients to access their logical volumes — file access protocol or block access protocol. Finally, we evaluate and compare those policies of stackable storage in order to unveil their design idiosyncrasies, performance behavior, and scalability.

Our proposed mechanism can provide the foundations for implementing centralized disk image management systems, and our evaluation is a first step towards understanding the intricacies of this emerging trend in systems management. The results of our analysis provide some guidance on how to exploit the convenience of centralized management without sacrificing efficiency and scalability.

The remainder of the paper is organized as follows. Section 2 enumerates the related work. In section 3 we describe the idea behind *stackable storage* and how it eases the management of disk images, whereas section 4 discusses the possible ways of implementing the mechanism of *stackable storage*. Section 5 evaluates stackable storage in terms of performance and scalability and, finally, section 7 draws our conclusions.

2 Related Work

The concepts of stackable storage are present in many file server software stacks, employed primarily as a mechanism to allow for temporal snapshots of the filesystems. Some systems, such as Linux's LVM [18], are architected to keep temporary snapshots to facilitate backup. Others such as NetApp's Write Anywere File Layout (WAFL) file system [7] and ext3cow [23] keep snapshots around for an extended period of time. Other approaches, such as the Plan 9 File Server [25] kept snapshots on a permanent basis by storing them on WORM optical jukeboxes. Later versions [28] leveraged content-addressable-storage [27] to accomplish the same purpose with more frequent snapshots. File systems such as Elephant [9] offer more granular versioning and allow different retention policies for historical versions. More recently, Peabody [17] provided time-traveling disks which expose a block-level interface and provide complete versioning of disk operations regardless of the file system using it.

Stackable file system design is a relatively old concept [31]. Within the Linux community, more complicated stackable file systems have been enabled via the File System Translator language (FiST) [39]. It has been used to implement a variety of file system extensions including encrypted file system layers [15] and compressed file system stacks. More relevant to our purposes, it was used to implement a versioning file system [19] which provided fine grain versioning of file modifications similar to CVFS [33]. FiST was also employed in the creation of Unionfs [26], which we use as the basis for our stackable file system evaluation.

Another form of file system based stackable storage was developed with the intent of coalescing duplicate files This can be found in the Windows 2000 server operating system in the form of their single instance storage components [5]. It searched through the file system for identical files, replacing copies with special links, and employed a copy on close mechanism to recitify changes made to instances of the file. The primary use of SIS was in support of thier Remote Install Server, allowing rapid deployment of new systems.

Stackable block storage can often be found in the service of virtualization technology. VMware [37], Xen [10], and even full-system emulators such as QEMU [4] use virtualized disks with the ability to use block-level copy on write mechanisms to share common base images. Virtualization aware file systems such as Ventana [24] attempt to merge aspects of stackable file systems with stackable block storage.

The increased popularity of scale-out server systems along with the desire to improve manageability of client systems has led to a large number of network install and provisioning management systems. These range from disk imaging systems such as Symantec Ghost [34] to more complicated solutions combined with provisioning products such as Tivoli's Provision Manager for OS Deployment [36]. Other imaging solutions, such as IBM Director's Remote Deployment Manager (RDM) [16] can also link to server storage provisioning tools (SSPT) [13] which can optimize disk imaging directly on the back-end storage server.

Another class of management software seeks to deploy systems in diskless and psuedo-diskless configurations using network file systems. Blade Fusion [1] provides products which allow rapid deployment to blades by dedicating one or two blades as management systems and dedicated file servers. "Golden" file system images are stored on the central file server, and cloned to dedicated file hiearchies which are then exported to individual blades. Other firms such as [2] provide wider area deployment and storage management solutions. Levanta provides the Intrepid M [3] Linux management appliance which provides a complete solution for rapid deployment of Linux systems using a combination of NFS and their own stackable file system solution called mapfs.

While there has been many recent studies [29] [35] [14] comparing SAN and NAS technologies – there were none who combined this analysis with evaluation of stackable storage technologies and tested in scale-out configurations. We seek to provide such an evaluation in section 5.

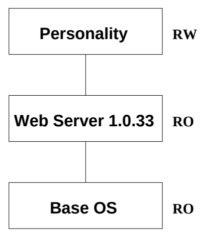

Figure 1: *Layers of a Web server logical volume.*

3 Stackable Storage for Systems Management

In this section, we describe our stackable storage mechanism in the context of centralized systems management, and discuss how this mechanism can be used to carry out management tasks on behalf of the system administrator.

3.1 Stackable storage

We refer to stackable storage as a mechanism to stack layers of disk image in order to compose different views of a centralized storage system, where each stack defines a logical volume on which a particular client machine relies. A storage server (or storage appliance) maintaining all layers exported to client machines is the single point of management.

Our management model is based on three types of disk image layers. On the bottom of all stacks is the *base operating system layer* which is shared by all client machines. It consists of the base file system that the operating system needs to function.

The second type of disk image layer is the *role-specific layer*. This one lies on top of the base operating system layer complementing it with the file system portion whose contents define the role (or the purpose) of the machine. The storage server might provide a number of predefined roles tailored according to the needs of a particular organization. For instance, in a typical multi-tier online Internet service, the roles available for deployment could be *Load Balancer*, *Web Server*, *Application Server*, and *Database Server*.

Finally, each machine has its private *personality layer* on the top of its stack. The purpose of this layer is to store all changes made to the contents of the logical volume of the machine in question by means of a CoW (Copy-on-Write) mechanism; as such, it is the only layer that gets modified throughout the lifetime of the logical volume. The contents of the role-specific and base operating system layers remain intact since they are supposed to be shared.

In our multi-tier Internet service example, the logical volume of a machine that has been assigned the role Web Server would comprise three layers, namely: a personality layer on top of the Web Server layer on top of the base operating system layer. This logical volume is illustrated in figure 1; on the right of each depicted layer is its status — read-only or read-write. The Web Server machine is oblivious to the fact that its logical volume is formed by three layers. Internally, however, the stackable storage mechanism guarantees that all changes made to the contents of the logical volume are confined in the personality layer, allowing the Web server layer to be shared by all machines that have been assigned that role, and the base layer to be shared by all machines. By merging the logical volume layers and properly taking into account the changes stored on the personality layer, the stackable storage mechanism ultimately gives the machine the illusion of a united file system on a typical storage device.

In addition to distinguishing and exporting multiple role-specific layers, the storage server can also associate versions with each layer.

3.2 Management tasks

The abstraction of stackable storage, although conceptually simple, is a powerful feature that facilitates common base and role-specific images sharing, enables the storage server to obtain instantaneous snapshots of all logical volumes as well as to quickly rollback to a previously taken snapshot, and makes it trivial to perform fast system reprovisioning through simple role re-assignments. In the next paragraphs we describe how stackable storage can be used to handle typical systems management tasks.

Machine installation. From the administrator perspective, installing a machine is as easy as selecting a role and instructing the storage server to apply the selected role. As a result of this action, the appropriate storage stack will be built and exported as a logical volume to the machine.

Reprovisioning. We refer to reprovisioning as changing the role of a machine. At the storage level, two steps are performed to carry out this action. First, the machine's current role-specific layer is replaced with the layer corresponding to the new role. Second, in order for the administrator to be able to rollback to the state of the machine before it was reprovisioned, the current personality layer is saved as part of the repository of checkpoints and a new personality layer is assigned to the machine. By saving the current personality layer the storage server can recreate the original stack if required by the administrator. Figure 2 depicts a Web Server being reprovisioned as a Database Server. On the left is the original storage stack that is replaced with the logical volume whose composition is shown on the right.

Checkpoint and rollback. To create a checkpoint of

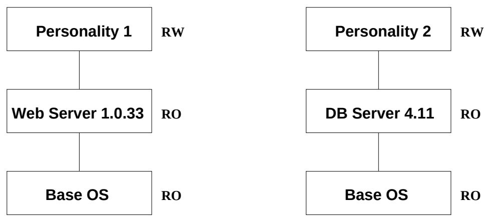

Figure 2: *Web Server is transformed into a Database Server.*

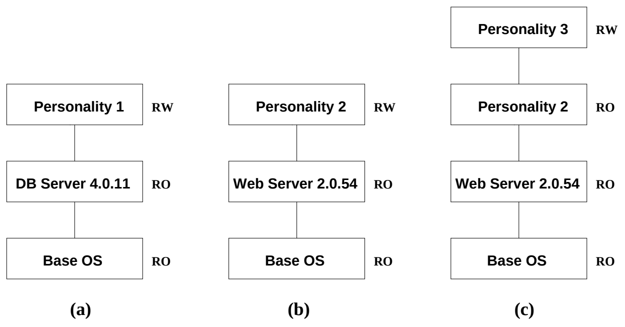

Figure 3: *Rollback operation: (a) stack before rollback; (b) stack corresponding to the snapshot chosen for rollback; (c) new logical volume created as a result of rollback.*

the logical volume of a given machine, the storage server performs three actions, namely: (1) it saves the current personality layer along with the stack configuration; (2) it transforms the current personality layer into a read-only layer; and (3) it adds a new personality layer onto the stack. Actions (2) and (3) guarantee that the changes made to the logical volume after the checkpoint was taken will be confined in a different personality layer. As a consequence, action (1) will save only the difference between the current state of the logical volume and the state at the time the previous checkpoint was created.

In rolling back a logical volume to a previous snapshot, the storage server first recreates the logical volume based on the stack configuration saved as part of the checkpoint. Next, it transforms the personality layer of the recreated stack into read-only and inserts a new personality layer on the top of the stack. Adding a new personality layer during a rollback ensures that all checkpointed states remain valid at all times.

To illustrate the rollback operation, let us suppose that a certain machine has its logical volume configured as shown in figure 3(a), and that the system administrator wishes to rollback to a snapshot corresponding to the stack shown in figure 3(b). The logical volume given to the machine as a result of the rollback operation will have the layering configuration depicted by figure 3(c). Note the new personality layer on top of the stack and the change of the status of the previous personality.

Machine cloning. Stackable storage allows the system administrator to easily clone a machine. To carry out such an action the storage server saves the personality layer of the machine to be cloned as a new personality layer, and creates a stack identical to that of the original machine, except that the copy of the original personality layer is used. The new stack is then exported as a logical volume to the clone.

Role publishing. The idea of role publishing is to extend the storage server by either adding new roles or creating upgrades for existing ones so that the new roles and upgrades are made available for future deployment. At the storage level, each new role or new version of a role translates into a new role-specific layer that can be selected to integrate logical volumes.

In our model of stackable storage, adding a new role can be done quite easily. To that end, the system administrator might use a machine whose logical volume is formed by only two layers: a clean personality layer on top of the base operating system layer. From such a machine, the administrator effectively installs the software intended to be the new role. Once the installation is complete, the administrator commands the storage server to capture the personality layer of the logical volume on which the installation was performed and to save it as a new role with a given name and version.

Although adding new roles is rather trivial, the creation of upgrades bears subtleties that are worth discussing. There are two ways of reasoning about upgrades, depending on the configuration of the logical volume from which they are derived. In its simplest form, a new version of a role (an upgrade) is derived in the same manner as a new role: from the base layer. Roles and upgrades so derived depend on the base layer; in other words, they will always be stacked on top of it.

For convenience of installation or space efficiency considerations, upgrades might be derived directly from the role to be upgraded, as opposed to from the base layer. To clarify this issue, let us suppose that the administrator wants to create an upgrade for the Web Server role. Instead of installing the new version from the scratch, i.e., from the base layer, the administrator might apply a patch to an already available older version of the role. To do so, a machine with a logical volume containing a clean personality layer on top of the layer corresponding to the older version of the Web Server role should be used. Once the patch has been applied, the administrator instructs the storage server to capture the personality layer of the logical volume and to save it as a new version of the Web Server role. Unlike the scenario described in the previous paragraph, upgrades derived in this fashion depend on the older version; therefore, the layer corresponding to the newly created upgrade must

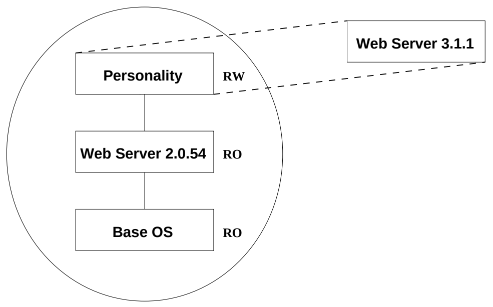

Figure 4: *Creation of an upgrade for the role Web Server. The personality layer is captured and saved as Web Server version 3.1.1.*

always be stacked on top of the layer containing the older version. Figure 4 depicts the creation of an upgrade for the Web Server role. The version 3.1.1 was derived from the version 2.0.54. All stacks where the layer Web Server 3.1.1 appears will have the layer Web Server 2.0.54 under it. Needless to say, in order to ensure that the stacks satisfy this kind of dependency rules, the storage server needs to record them.

Upgrade deployment. Once an upgrade is created and published as discussed above, it becomes available for deployment. Similarly to upgrade creation, there are two different approaches to upgrade deployment. One possibility is to treat it as a reproviosining, in which case the personality layer of the original logical volume is not kept in the resulting stack, as illustrated in figure 2. The assumption here is that the contents of the personality layer are not relevant in the context of the new stack configuration. Although this rationale makes sense when the role of a machine is changed, it may be unreasonable when the role remains the same. To accommodate the situations where it is desirable to maintain the current personality layer, the stack resulting from an upgrade deployment would comprise the base layer, the layer corresponding to the upgrade (observing the dependency rules), the original personality layer with read-only status, and a new writable personality layer at the top. An example of this scenario is shown in figure 5, where the upgrade created in figure 4 is effectively deployed. Note that the former personality layer is part of the resulting stack.

It should be clear that it is possible for a stack to have more than two personality layers. This happens, for instance, when an upgrade is deployed on a logical volume that has more than one personality layer. As a consequence of this multitude of personality layers, dependency rules should apply to them. For example, in figure 5, *Personality 2* depends on *Personality 1*.

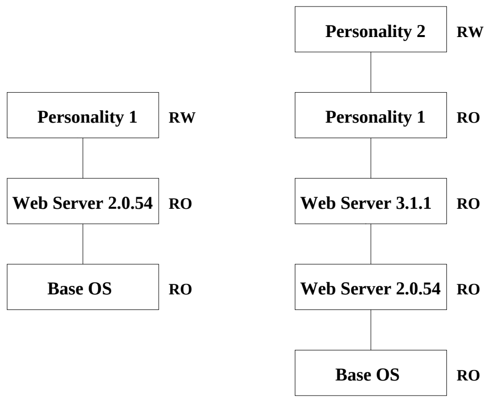

Figure 5: *Role Web Server is upgraded from version 2.0.54 to 3.1.1.*

4 Stackable Storage Implementation

In evaluating our stackable storage abstraction, we identified three dimensions in the design space. One dimension is *stack granularity*, i.e., the granularity at which the disk image layers are created and stacked. The possible options here are to manage layers either at the file system level or at the block level.

Another important aspect to consider is *stack location*. Although our approach to disk image manipulation relies on data stored on a centralized server or storage appliance, the code that implements the stacks can run either on the server or on the clients.

Finally, *data transport protocol* is yet another facet of stackable storage, the options being to employ a file transport protocol (e.g., NFS) or a block transport protocol (e.g., iSCSI). To appreciate the importance of data transport protocol as a component of the design space, one should realize that *stack granularity* does not necessarily dictate a particular type of protocol. Clients may access data by means of a block transport protocol even if the layers are managed at the file system level; the converse is also true.

In this section, we discuss all possible combinations in the design space, grouping them by *stack granularity*. We first delve into stacking at the file system level and, in the sequel, describe block-level stacking.

4.1 File system stacking

File systems that support the notion of namespace unification, such as UnionFS [38], can be used to implement stackable storage. In this work we leverage UnionFS not only due to its widespread use, but also because the level of abstraction it offers is well suitable for our conception of stackable storage.

UnionFS was designed for Linux kernel series 2.4 and 2.6. It is actually a piece of software lying between the

Linux VFS (Virtual File System) and the lower-level file systems (e.g., ext2, ext3, ReiserFS, etc.) that captures calls made to VFS and modifies the behavior of the corresponding operations in order to implement the functionality of namespace unification.

The main idea behind namespace unification is to merge different directory subtrees (possibly belonging to different file systems) into a single unified view. This can be done recursively for subdirectories (deep unification) or just for a one level of directories. The latter is more efficient but it does not provide for our needs. UnionFS does deep directory unification. In UnionFS terminology, each merged subtree is called a *branch*, and the resulting merged file system is called a *union*. We treat each merged branch as a layer, and the resulting union as a stack. In our stackable storage server, the base operating system layer, the role-specific layers, and the personality layers are separate branches. When building a stack, the storage server merges the appropriate layers using UnionFS in such a way that each layer has higher precedence than the one located under it. This is of paramount importance to ensure that the top-most personality layer — the only one that gets modified — is always the most influential in the logical volume contents perceived by the client. Hence, if a file gets deleted, a whiteout (a "negative file") is created on the personality layer indicating that the corresponding file should not appear on the logical volume; if a file gets modified, the version of the file stored on the personality layer is the one accessible from the logical volume; if a new file is created, it will be present only on the top-most personality layer and, therefore, will naturally show up on the logical volume.

In the realm of stackable storage based on file system stacking, there are three design options: server-side stacking with file transport protocol; client-side stacking with file transport protocol; and client-side stacking with block transport protocol. The fourth combination, server-side stacking with block transport protocol, is unfeasible. In the next paragraphs we describe each combination.

Sever-side stacking and file transport protocol. In this configuration, the storage server effectively merges the layers and exports the stacks to the clients. Our prototype uses UnionFS for namespace unification, as already mentioned, and NFS is the file transport protocol we explored, due to its popularity. To make a given stack available to a particular client, the storage server builds it, mounts it on the local file system, and exports its mount point through NFS to the appropriate client.

Client-side stacking and file transport protocol. This approach differs from the previous one in that the storage server does not export the stacks to the clients; instead, it exports the layers and the clients build their stacks by running UnionFS locally. It is up to the clients to mount through NFS all layers required to build their stacks.

Client-side stacking enables two policies for managing the private personality layers. A client can either access its personality layer via NFS or keep it locally. The latter has the potential to increase performance by providing some level of caching.

Client-side stacking and block transport protocol. In our prototype we used iSCSI as the block transport protocol. Like NFS, iSCSI enables the access to storage servers by means of IP-based networks. Differently, however, while an NFS server exports directories to clients, an iSCSI server makes a portion (or all) of its local disk(s) available to clients, which, in turn, operate on the remote disk(s) in exactly the same way as they would use a local disk. In particular, a Linux iSCSI server exports block devices to clients, where a block device might be a disk partition — e.g., `/dev/sda5`, `/dev/sda6`, etc.

Block transport protocol and file system stacking can coexist, provided that the stacking is done at the client side and the block devices exported by the storage server are not shared for writing. To satisfy the latter requirement, the server maintains one block device containing all shared layers (base operating system layer and all role-specific layers) and exports it to the clients. Since these layers represent the read-only portion of all stacks, the clients can safely import the block device and access it through iSCSI.

Regarding the personality layers, there are two options for managing them. Either the storage server provides one block device per personality layer, or the clients keep their personality layers locally. Local personality layers can potentially lead to overall better performance since they would provide some level of caching.

4.2 Block stacking

An alternative to stacking directory subtrees by means of namespace unification is to build the stacks at the level of disk blocks. This variation of stackable storage is implemented at a lower level of abstraction, with no file system involvement.

In Linux, the target platform of our prototype, block-level stacking can be accomplished by treating block devices (e.g., `/dev/sda5`) as layers of the stacks. In this case, since two or more block devices are part of a stack, stacks are treated as logical block devices which, in turn, represent the logical volumes exported to the clients. This idea introduces the concept of *stackable block devices*.

Linux LVM (Logical Volume Manager) [18] and EVMS (Enterprise Volume Management System) [11] could potentially be used to implement *stackable block devices*, as they support a copy-on-write mechanism for taking snapshots of logical volumes. They were conceived as a backup mechanism that freezes the file system for a short period of time backing up the blocks as they are modified. To implement stackable storage, we only need to write to the right level of the stack and update the metadata, which is much more efficient than waiting for a copy to complete the write. Additionaly, the metadata used for block addressing is not appropriate for situations where a huge number of blocks are modified. We measured the performance of writing to a LVM snapshot using bonnie++ locally and found that the performance degradation compared to the performance of a non stacked block device was of an order of magnitude.

For the reasons stated above, we implemented our own block stacking driver for Linux kernel 2.6 to realize the idea of *stackable block devices*. Our driver is used by the device-mapper [8], a component of the Linux kernel that creates logical block devices by means of published drivers called *targets* in the device-mapper terminology. Each logical block device created by the device-mapper by means of our *target* is actually a stack of block devices that the storage server exports as a logical volume to a certain client.

Associated with each stack is a bitmap that indicates on which layer each block resides. For instance, the bitmap may indicate that block 10 is on the personality layer and block 90 is on the base operating system layer. A newly created logical volume has its personality layer empty; as the contents of blocks change, they are written to the personality layer and the bitmap is updated accordingly. This provides with a one to one block static mapping which wastes space. There is a compromise between incurring in the cost associated with sparse allocation and wasting space. We have left the sparse allocation out of the picture on purpose to isolate the variables of the study.

Stackable storage at the granularity of blocks lends itself to two combinations of *stack location* and *data transport protocol*, as described in the next few paragraphs.

Server-side stacking and block transport protocol. In this configuration, the storage server stacks the block devices and exports the resulting logical volumes to the clients. Our prototype, as mentioned before, adopts iSCSI as the block transport protocol. Therefore, the logical block devices created by the storage server are exported through iSCSI to the clients.

Client-side stacking and block transport protocol. In this variation of block stacking, the storage server exports through iSCSI the block devices to be stacked (the layers). The clients then import the block devices they need and build their stacks.

Stacking on the client side allows each client to keep the block device representing its personality layer locally, as opposed to having the server export it. This might improve the overall data access performance thanks to caching.

5 Evaluation

This section evaluates the different flavors of stackable storage we previously described. In particular, we compare them with respect to performance and scalability.

5.1 Experimental setup

We conducted our experiments on nine blade machines. Each machine is equipped with:

- two dual-core 2.4 GHz AMD Opteron processors, with 1024 KB of L2 cache per core;
- 4 GB of memory in 4 DIMMS of 1GB DDR RDIMM memory PC3200R;
- a Broadcom NetXtreme BCM5704S Gigabit Ethernet card.

We used one of the blades as the storage server, and the remaining eight as clients. The server has:

- two 73 GB ST973401LC SCSI disks operating at 10,000 RPM and average seek time of 4.1ms, with an ultra320 SCSI interface;
- an LSI53C1010R Ultra160 SCSI controller.

The server blade runs the Linux Ubuntu Dapper distribution with the latest updates applied and kernel 2.6.17. All client blades are diskless and boot from a separate server that provides them with NFS-roots based on the Red Hat Enterprise Linux distribution version 4 and kernel 2.6.17. All 9 blades and the NFS-root server are connected to the same Gigabit Ethernet switch.

All experiments using NFS as the file transport protocol rely on NFSv3 mounted asynchronously. Regarding iSCSI, the storage server runs iSCSI Enterprise Target version 0.4.13 and the clients use Open-iSCSI version 1.0-485 as the iSCSI initiators. TCP is the transport protocol used for both NFS and iSCSI. For implementing the stackable file system, we use the CVS snapshot “20060616-1848” of UnionFS version 1.2. *Ext2* is the file system adopted to form the layers of our UnionFS-based stacks and to format the block devices used with iSCSI. One of the disks of the storage server is dedicated to our experiments, while the other is used only by the operating system.

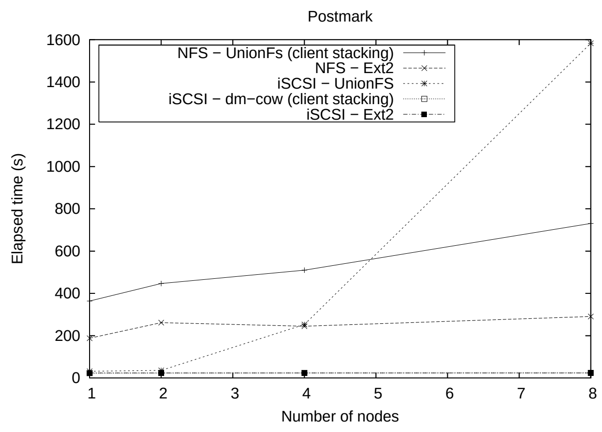

Figure 6: *Postmark results for client-side stacking.*

In terms of stack composition, our evaluation takes place on a three-layer configuration comprising the base operating system image, a role-specific layer (Apache Web Server image), and the personality layer. All disk read operations are satisfied with data residing on the personality layer so that we can isolate and properly compare the overhead of layer lookup of both file system and block stacking. Therefore, our numbers do not account for the time it takes to copy a file from a read-only layer to the personality; this is done by the file system stacking approach whenever a file not present in the personality layer is modified.

In order to evaluate the performance and scalability of our different stackable storage configurations, we chose two widely used benchmarks: Postmark and Bonnie++. Since the former is metadata-intensive and the latter is data-intensive, our evaluation covers these two fundamentally different cases. In the sequel, we present the results of our experimental evaluation.

5.2 Postmark results

The Postmark benchmark first creates an initial pool of files of varying sizes. After creating the pool, it enters the so-called transaction phase, during which it randomly chooses one of two types of file operations to perform: (1) create or delete a file; (2) read a file or append data to a file. In our Postmark runs, we configured it to generate a pool of 20,000 files, whose sizes range from 500 to 1,024 bytes, and to perform a total of 200,000 operations. The two types (and subtypes) of file operations were configured to be selected with equal probability. This workload mimics the use of multiple small short-lived files, which is common in applications such as online web-based e-commerce, electronic mail, news, etc.

Figure 6 shows the runtime of Postmark for three stackable storage configurations: block stacking with block transport protocol (iSCSI-dm-cow), file system stacking with block transport protocol (iSCSI-UnionFS), and file system stacking with file transport protocol (NFS-UnionFS). In all cases, the stack is built at the client side. The graph also shows the baseline for iSCSI (iSCSI-Ext2) and NFS (NFS-Ext2). The plotted data points represent runs with 1, 2, 4, and 8 client machines.

Comparing the baseline results for the two different transports, we see that iSCSI performs roughly an order of magnitude better than NFS for meta-data intensive workloads such as Postmark. This is the result of NFS sending all meta-data operations over the network to be resolved by the file system executing on the server. As a result, the server disk is already operating at 75 percent of its maximum number of transfers per second, even in the single client case. By contrast, in the iSCSI case, file system operations are resolved locally – primarily within the system's cache. Since iSCSI volumes are implicitly exclusive, no coherence traffic is required allowing for very aggressive cacheing of both meta-data and file contents.

When UnionFS is used to combine two NFS mounts it increases the number of meta-data operations performed due to additional lookups in the multiple layers. These additional operations are enough to exceed the maximum transactions per second the disk is capable of and result in the doubling of the Postmark run-time.

When UnionFS is used to combine two iSCSI mounts, it does much better for one or two clients – approximating the performance of the iSCSI base case. However, starting around 4 clients the additional meta-data operations from UnionFS start to disrupt the iSCSI cache resulting in a dramatic increase in overhead. This problem becomes much worse in the 8-node case, exceeding the overhead of UnionFS on NFS.

By contrast, when block stacking is used to combine two iSCSI volumes there is no noticeable overhead for all runs — the curves labeled iSCSI-Ext2 and iSCSI-dm-cow coincide. This is primarily due to the relatively small footprint of Postmark allowing the entire bitmap for the block stack to be cached in memory. This results in a very low latency for resolving the layer to load blocks from. This combined with iSCSI's aggressive cacheing of meta-data and content yield excellent results for meta-data intensive benchmarks.

Figure 7 is similar to figure 6, except that it presents the results for server-side stacking. iSCSI-UnionFS is not present on the graph because server-side stacking is not applicable. Following the same trend as in client-side stacking, iSCSI-dm-cow does not suffer from any visible overhead for the same reasons stated before.

The behavior of the configuration NFS-UnionFS is not noticably affected by stack location when up to 8 clients are used, as can be seen by comparing figures 6 and 7.

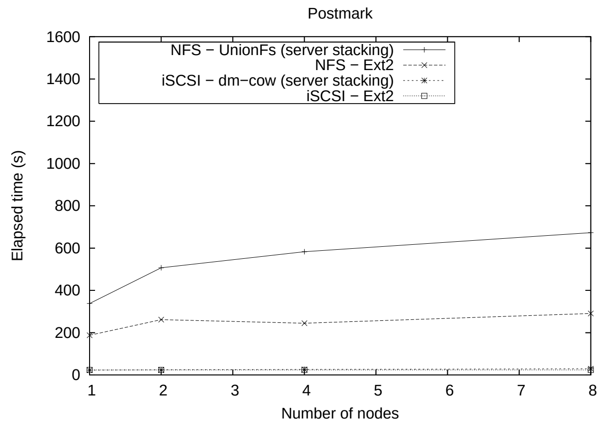

Figure 7: *Postmark results for server-side stacking.*

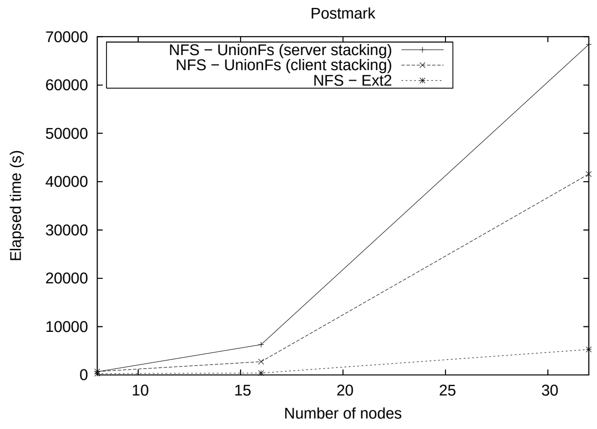

Figure 8: *Benefit of client-side stacking.*

We were a bit surprised by this result as we had thought the extra processor and memory overhead of UnionFS would bottleneck a server-side stack. To investigate further, we conducted Postmark runs with 16 and 32 clients. Since we had only 8 blades acting as clients, we ran 2 and 4 instances of Postmark per machine to mimic 16 and 32 clients, respectively. The results of these experiments are shown in figure 8. Matching our intuition, offloading the file system stacking code to the clients is beneficial when multiple clients are supported under metadata-intensive workloads – but will come at the cost of processor power and memory on each client.

We were not able to run the large-scale experiments with iSCSI as they would require 16 and 32 different block devices for the personality layers, i.e., 16 and 32 partitions. Unfortunately, Linux does not support more than 15 partitions per SCSI disk, and we did not have room in our server blade for more than a single SCSI disk (we did not want to use the root disk of the blade as we felt it would complicate analysis). Having multiple instances of Postmark running on the same node share a mount point for the personality layer is not an option either, since local caching effects would render the results uncomparable to the previous ones.

5.3 Bonnie++ results

The Bonnie++ benchmark creates a file of a specified size and performs a number of character-based and block-based operations on it. It is meant to mimic database workloads. In our experiments, we explored the block-based operations. Bonnie++ has three distinct phases: (1) file creation by means of a sequential write; (2) file rewrite, during which each block is read, dirtied, and rewritten, requiring a seek operation; and (3) file read, when the whole file is sequentially read. In order for the results to be valid, the size of the file is recommended to be twice the size of the memory.

Figures 9, 10, and 11 show the throughput in Kblocks/s for each of the Bonnie++ phases: write, rewrite, and read, respectively. In the x-axis of each graph we grouped the results by the number of clients used during the experiment.

The main limiting factor for the performance obtained with Bonnie++ is the seek time of the disk when multiple clients are used. The sub-linear behavior shown in the bar charts as the number of clients increases is mainly due to this effect: the locality is perturbated by multiple running instances of Bonnie++. This effect accentuates the cost of fetching the metadata blocks between contiguous requests the first time they are used, because some contiguous blocks are not sent together to the disk due to the extra seeks being generated by the breakdown of locality.

Bonnie++ evaluation makes a more interesting case for analyzing block storage. Local block stacking writes have much worse Bonnie++ throughput for one and two clients. The bitmap blocks are not yet in the cache, so they have to be fetched. The latency of the network plus the fetching of the bitmap blocks is enough to break up some requests and to generate extra seeks. Both effects combined explain the bad write behavior. In the rewrite and the read test, some of the bitmap blocks have already been exercised. This means they are in the cache. The bitmap blocks from the shared device on the server are prefetched by other clients. This explains the different behaviors for one and two clients. As the number of clients grows, the locality diminishes and almost every request costs the same, because it generates a new seek. The consequence of this is that metadata fetches do not matter anymore because they do not add any extra seek and all the curves converge to roughly the same point.

For the file system stacking case, note that client-side stacking performs better for rewrites and reads. A read

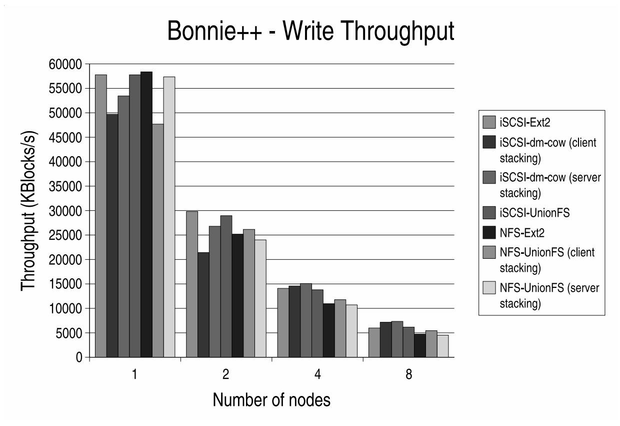

Figure 9: *Bonnie++ write phase.*

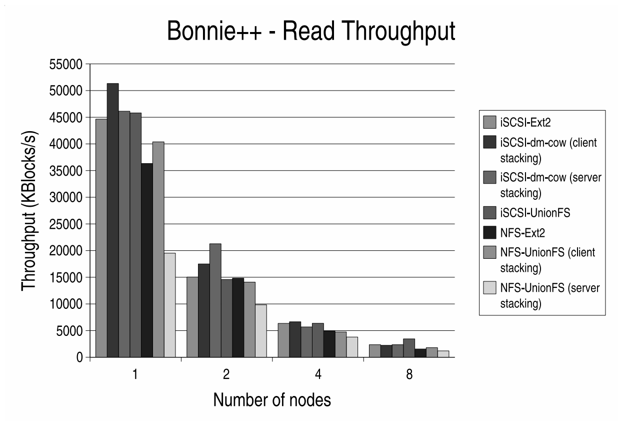

Figure 11: *Bonnie++ read phase.*

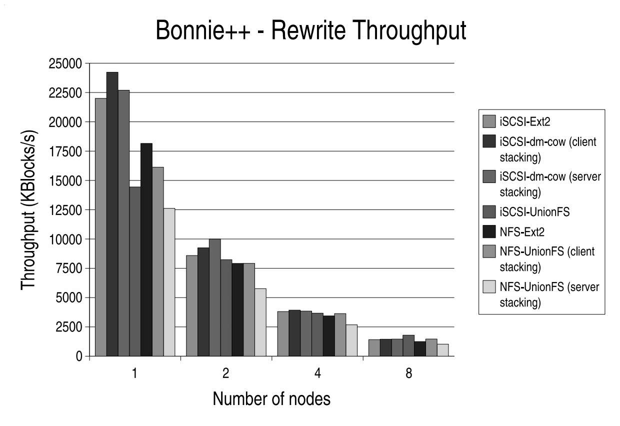

Figure 10: *Bonnie++ rewrite phase.*

followed by a write over NFS generates extra metadata fetching, and this situation is aggravates a server-side UnionFS stack. In the case of reads, the extra seeks due to fetching metadata causes a similar problem, also aggravated with UnionFS extra lookups sent over NFS. For client-side stacking this multiplicitive effect of extra lookups due to UnionFS plus NFS does not happen. Since each layer is NFS-mounted on the client, extra metadata operations generated because of NFS are not accentuated by the extra lookups performed by UnionFS.

6 Future Work

Most of the performance penalties we see while using NFS are due to extra meta-data traffic being transmitted in order to give NFS loose consistency. This results in giving iSCSI a distinct advantage in that it makes no attempt to maintain any sort of consistency. It would be interesting to repeat the evaluation with NFS configured to aggressively cache meta-data and content. Studying the use of client disk caches for both NFS and iSCSI to further hide any stacking overhead would also be interesting. Comparing stacking on NFSv3 with NFSv4 and other file system protocols such as AFS, CIFS, or various cluster file systems would further broaden the coverage of this study. Similarly, we'd like to compare stacking on iSCSI with stacking on ATA over Ethernet, Network Block Devices, or other SAN transports.

The performance analysis we conducted only accounted for two layer stacks. The overhead of multi-layer UnionFS stacks has been measured before [38], but no similar analysis has been done for block-stacking. Additionally, while we measured performance, we did not compare the storage efficiency of the two methodologies.

Two shortcomings in our evaluation methodology were the use of only a single target disk and the lack of a benchmark which operated on more than just the write-layer of the stacks. While both of these choices were made in supported our primary target application, a broader study covering the use of multiple disks, perhaps in a RAID configuration – and the use of a benchmark which stressed accesses to multiple layers of the stack would provide a more complete picture.

One problem of filesystem stacking which did not manifest itself due to the nature of our benchmarks is the relatively large penalty for copy on write with large files. In file system stacking such as UnionFS and mapfs, even a single character change results in a copy of the entire file. This type of penalty is likely unacceptable for systems, such as database servers, where large files are predominant. Use of a benchmark which better demonstrated the effect of this overhead may have painted a different picture for the stackable file system configurations.

While Bonnie++ and Postmark are broadly used benchmarks, it would have been nice to see the impact of stacking on large-scale, real-world applications over an extended period of time. We are in the process of deploying stackable storage within our data center and installing instrumentation which will allow us to collect this information.

7 Conclusions

The results clearly seem to favor block-stacking and iSCSI. It has a similar throughput to file-stacking for the Bonnie++ benchmark, but shows clear advantages for Postmark. As reported in the evaluation section this is primarily due to block-stacking's low overhead for meta-data operations and iSCSI's more aggressive use of the cache.

The interpretation of both the results for Bonnie++ and Postmark also suggest a case for performing the stacking operation on the clients. In the Bonnie++ case, the negative performance introduced by local stacking is negligible for four clients or more because requests are already being broken and seeks generated by the interleaving of different clients. By using local stacking, we can offload a good deal of cpu and memory usage from the server at a very low performance penalty. This will ultimately lead to better performance at even larger scales.

A real advantage to filesystem based stacks is that their duplication of meta-data gives them a higher degree of reliability. Failure of lower levels of a block-stack breaks all the levels above it. Filesystem stacking technologies do not have this problem – although most do not allow modification of lower layers once a new top layer is in place. While reliability problems may be mitigated by backups, RAID, and mirroring of disks, it remains a concern with block-stacking based approaches.

Some authors [24] have made a case for the manageability of filesystem based stacking due to their easy navigation and manipulation with normal file tools. In block stacking technologies, each layer of the stack is dependent on the layers beneath it and cannot be mounted independently. However, our experiences with Plan 9's Venti [27] have shown us that with appropriate tooling and minimal meta-data, dealing with multiple block-based stacks can be as simple as dealing with file-based stacks. Exploring the potential of hybrid stacking technologies which leverage the performance of block-based stacking with the flexibility of file-system based stacking remain an exciting area of future research.

Image-based deployment and maintenance of servers seems to be an increasing trend in the industry. As low-cost servers and blades drive the popularity of scale-out systems, efficient and centralized management will become more and more critical. The multiplicitive effect of server virtualization on the number of system images will further intensify its importance. We believe development of efficient, flexible, and scalable storage technology is the key to this centralized management. Stackable storage techniques improve the efficiency and effectiveness of image deployment and maintenance. They also enable rapid sandboxing, cloning, redeployment, and rollback of systems to meet the demands of today and tomorrow's service oriented architectures.

References

[1] Bladefusion technolgoies. http://www.bladefusion.com.

[2] Corente. http://www.corente.com.

[3] Levanta intrepid m. http://www.levanta.com/, 2005.

[4] BELLARD, F. Qemu, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference* (2005).

[5] BOLOSKY, W. J., CORBIN, S., GOEBEL, D., AND DOUCEUR, J. R. Single instance storage in windows 2000. *4th USENIX Windows Systems Symposium* (2000).

[6] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., AND LAM, M. S. The collective: A cache-based system management architecture. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation (NSDI '05)* (May 2005).

[7] DAVE HITZ, E. A. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter Conference* (January 1994).

[8] Linux device-mapper. http://sourceware.org/dm/.

[9] DOUGLAS S. SANTY, E. A. Deciding when to forget in the elephant file system. In *Proceedings of the seventeenth ACM symposium on Operating Systems principals* (1999).

[10] DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., PRATT, I., WARFIELD, A., BARHAM, P., AND NEUGEBAUER, R. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2003).

[11] Enterprise volume management system. http://evms.sourceforge.net/.

[12] GRAY, J. Why do Computers Stop and What Can Be Done About It? In *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems* (Jan. 1986).

[13] GREG PRUETT, E. A. Bladecenter systems management software. *IBM Journal of Research and Development* (November 2005).

[14] GULABANI, S. Aix performance with nfs, iscsi and fcp using an oracle database on netapp storage. TR-3408, 2006.

[15] HALCROW, M. A. ecryptfs: An enterprise-class encrypted filesystem for linux. *Proceedings of the Ottawa Linux Symposium* (1996).

[16] IBM. Ibm directory remote deployment manager. http://www.ibm.com, 2006.

[17] III, C. B. M., AND GRUNWALD, D. Peabody: The time travelling disk. *mss 00* (2003), 241.

[18] Logical volume manager. http://sources.redhat.com/lvm2/.

[19] MUNISWAMY-REDDY, K.-K. VERSIONFS: A versitile and user-oriented versioning file system. Master's thesis, December 2003.

[20] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)* (Dec. 2004).

[21] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. Why do Internet Services Fail, and What Can Be Done About It. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'03)* (Mar. 2003).

[22] PATTERSON, D. A., BROWN, A., BROADWELL, P., CANDEA, G., CHEN, M., CUTLER, J., ENRIQUEZ, P., FOX, A., KICIMAN, E., MERZBACHER, M., OPPENHEIMER, D., SASTRY, N., TETZLAFF, W., TRAUPMAN, J., AND TREUHAFT, N. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, University of California, Berkeley, Mar. 2002.

[23] PETERSON, Z., AND BURNS, R. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage 1*, 2 (2005), 190–212.

[24] PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the 2006 Network Systems Design and Implementation conference* (2006).

[25] PIKE, R., PRESOTTO, D., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. Plan 9 from Bell Labs. *Computing Systems 8*, 3 (Summer 1995), 221–254.

[26] PUJA GUPTA HARIKESAVAN, E. A. Versatility and unix semantics in a fan-out unification file system.

[27] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies* (Monterey,CA, 2002).

[28] QUINLAN, S., MCKIE, J., AND COX, R. Fossil, an archival file server.

[29] RADKOV, P., YIN, L., GOYAL, P., SARKAR, P., AND SHENOY, P. A Performance Comparison of NFS and iSCSI for IP-networked Storage. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST '04)* (Mar. 2004).

[30] Rembo technologies. http://www.rembo.com/.

[31] ROSENTHAL, D. Evolving the vnode interface. *Proceedings of the Summer USENIX Technical Conference* (1990).

[32] SAPUNTZAKIS, C., BRUMLEY, D., CHANDRA, R., ZELDOVICH, N., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA '03)* (October 2003).

[33] SOULES, C., GOODSON, G., STRUNK, J., AND GANGER, G. Metadata efficiency in versioning file systems, 2003.

[34] SYMANTEC. Put imaging to work for you. http://sea.symantec.com/content/article.cfm?aid=99, 2004.

[35] TECHNOMAGES, I. Perforamnce comparison of iscsi and nfs ip storage protocols. http://www.technomagesinc.com.

[36] TIVOLI. Tivoli provisioning manager for os deployment. http://www.rembo.com/index.html, 2006.

[37] VENKITACHALAM, G., AND LIM, B. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference* (2001).

[38] WRIGHT, C. P., DAVE, J., GUPTA, P., KRISHNAN, H., QUIGLEY, D. P., ZADOK, E., AND ZUBAIR, M. N. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS) 1*, 4 (November 2005).

[39] ZADOK, E. Fist: A system for stackable file system code generation, 2001.