O2S2: Enhanced Object-based Virtualized Storage

Himanshu Raj
College of Computing
Georgia Tech
Atlanta, GA
rhim@cc.gatech.edu

Karsten Schwan
College of Computing
Georgia Tech
Atlanta, GA
schwan@cc.gatech.edu

Abstract

Object based storage devices (OSDs) elevate the level of abstraction presented to clients, thereby permitting them to offer methods for managing, sharing, and securing information that go beyond those offered by block-based stores. The Object-Oriented Storage System (O2S2) architecture presented and evaluated in this paper implements a virtualization service to provide object-based storage in a virtualized environment. This service provides a virtual object-based storage device (vOSD) to virtual machines. The use of vOSDs permits the service provider, i.e., the *vOSD storage domain*, to offer to guest virtual machines new methods for resource management and consolidation, without requiring the purchase of physical storage devices that faithfully implement OSD functionality. Methods demonstrated in this paper include improved support for access control and for heterogeneity of storage devices. Advantages derived from such methods also include reduced complexity for end clients, i.e., guest VMs. A prototype PVFS-based O2S2 implementation demonstrates that its enhanced services can be provided at low cost, enabled in part by the efficient utilization of otherwise idle domain resources.

1 Introduction

Storage virtualization is a mature area of computing, including commercial solutions, such as IBM's System Storage DS8000 and EMC's Centera. Such 'storage appliances' are in common use in well-networked environments like data centers, but low cost implementations have even enabled them for personal/home systems [4]. Their realizations utilize technologies like Network Attached Storage (*NAS*) and Storage Area Networks (*SANs*) to provide end clients with virtualized storage devices, where NAS and SAN technologies differ in the interfaces they provide to the end client. A SAN solution provides low-level block-based storage access, while a NAS solution provides higher-level file-based access. Abstracting from these differences and for simplicity, when referring to NAS or SAN, this paper terms the entity implementing any such virtualized storage solution a *storage domain*.

Any storage domain must answer multiple questions, including (1) what is the access interface provided to the client – block based vs. object (file) based, and (2) how is the data stored on (mapped to) physical devices? Depending on the answer to (1), the storage domain has different degrees of freedom concerning how to store the data. For example, if the interface is block-level, any storage decision must be made at that granularity, which also means that the client has the obligation to make such decisions, thereby increasing client complexity and reducing flexibility for the storage domain. Further, because of the large overheads of maintaining metadata about every single block, the storage domain is typically agnostic of the properties of the actual data stored, also implying that useful device properties like fault tolerance and striped I/O must be realized at the virtual block device granularity. In contrast, improved solutions are possible when allowing properties to be maintained on a per object basis, where an object-based interface provides opportunities for new reliability methods [57], for increased scalability [58, 24], for increased resource consolidation, and for data sharing among multiple clients [25].

Following the paradigm of object based storage, this paper presents the design and implementation of an Object-Oriented Storage System (O2S2) architecture that can provide virtual object-store devices (vOSDs) and services to any virtual machine via the *vOSD storage domain*, without the need for specialized object storage hardware [7]. Moreover, these vOSDs can provide *semantically enhanced – logical* – object storage. That is, for any object stored by a client on a vOSD, the latter can store additional attributes (i.e., metadata) such as provenance [11], consistency [38], and client-specific information pertaining to the semantics of its content (e.g., an object containing a 'health' record or one that contains multimedia data). Accordingly, some of these vOSD object attributes will be solely managed by the vOSD storage domain, in a client-oblivious manner, while others are shared between the clients and the domain. Regardless of how such management is performed, however, it is these attributes, along with the attributes of the physical devices being managed by the vOSD storage domain, that permit the domain to provide enhanced functionality and services to storage clients. In contrast, more traditionally, a vOSD stores raw data like that associated with files in a filesystem and limited semantic meta-data associated with these files, such as primitive access control information, size, type of data, and useful timestamps.

Beyond presenting the O2S2 architecture and its prototype implementation, our research also explores new and useful functionality associated with vOSDs. One class of such functionality, focused on the way objects are stored, concerns exploiting the different properties of physical storage media. The idea is that by aggregating an ensemble of physical devices, a storage domain can often provide better performance or enhanced functionality to the virtual device than by using a single physical device [48]. In addition, semantic information about the objects being stored can be used when aggregating physical devices. For example, an object-based file store may provide striping and RAID functionalities only to the objects that actually require them. As another example, the mapping between an object and a particular physical device can be decided based on object or device attributes, such as those pertaining to privacy and mobility constraints. In fact, such *semantic aggregation* is not unique to storage devices. Its use with camera devices, for instance, makes it possible to seamlessly join video streams from multiple cameras in order to provide a virtual video wall [53]. We note that semantic aggregation cannot be done with SAN solutions or with lower level device aggregation like that provided by the Logical Volume Manager [10]. This is because it is the object based interface and the object attributes that enable semantic aggregation at object granularity.

Another class of functionality enabled by the O2S2 architecture is fine-grained, object-based, access control. Based on the *labels* of objects and the clients who access them, the vOSD storage domain implements *Role-based Access Control* (RBAC) [26]. Enforcing access control at object granularity provides sharing and consolidation of resources superior to that offered by the large storage partitions present in block-based systems. Further, the storage domain can be integrated with the *trust management* component of a platform, where it can utilize 'trust' related information about a client to enforce *dynamic* RBAC. Additionally, such access control enables object-level logging of a client's accesses, which can be used to enhance security.

An important element of the O2S2 architecture is its use of a 'storage domain'. This provides independence from specialized storage hardware, such as object stores [7] and other enhanced disk-controllers [60]. In addition, the object properties implemented by a storage domain and desired by end users can transcend what is offered by backend hardware. Examples include encryption or secret sharing [49] techniques and transformation of data in client-specific ways [34], such as for obfuscation and specialization [61] purposes.

Finally, while vOSD storage domains can be constructed in many ways, this paper describes domain realizations geared to meet the challenges of virtualized execution environments. In this context, vOSD storage domain clients are *Virtual Machines* (VMs), which execute on a Virtualized Platform provided by a hypervisor or Virtual Machine Monitor (VMM). Additional Service VMs implement virtualized services for these VMs. The vOSD storage domain implements the storage service inside a Service VM; other examples include 'Dom0' providing network virtualization [43] and the VMedia runtime [45] providing multimedia device virtualization. Experimental evaluations presented in this paper, therefore, are carried out in the contexts of VMs, VM usage of the vOSD storage domain, and the VM-level overheads experienced in these settings.

Experimental results based on a PVFS-based prototype implementation of O2S2 architecture and its realization of vOSDs demonstrate (1) that the cost imposed by enhanced vOSD functionalities is low and (2) that such functionality scales well with an increasing number of client VMs. In particular, the cost of per-object access control is 2.5% and .4% for large reads and large writes, respectively, as compared to the case where no access control is enforced. Also, by using the resources available at the vOSD storage domain, it is possible to obtain performance benefits of $\sim 3X$ for large reads, as compared to current virtual block based storage solutions in the Xen virtualization environment.

2 Motivation

This section describes the scenarios that motivate the design of the O2S2 enhanced storage architecture. It describes how this architecture enables improved performance, better resource consolidation, easier trust management, and increased usability in heterogeneous storage environments.

2.1 Object-based Storage Interfaces

Object-based storage interfaces, like those presented by the file-based interfaces of cluster or distributed file systems (e.g., PVFS [21], LWFS [38], Lustre [6] and Coda [19]), and by object storage devices (OSDs) [56, 27, 25], provide notable benefits for the storage client. For instance, when the tasks of storage allocation and access control are delegated to the storage domain, this simplifies the client's kernel in that it is only required to run a minimal file system. Further, since operating systems already maintain information, both data and meta-data, grouped at file-level, object-based storage interfaces provide an appropriate match between the capabilities of the virtual device and the requirements of the client. Figure 1, derived from [56], highlights the differences between an object- and a block-based interface. vOSDs and the O2S2 architecture underlying them exploit these differences to maintain and use novel meta-data with storage objects, as explained in Section 2.2.1 below.

The vOSD storage domain benefits from the presence of object based interfaces, because storage management can be performed at a semantically meaningful level. This facilitates sharing and presenting opportunities for resource consolidation, and more importantly, it provides opportunities for *enhanced* virtual storage, based on additional per-object meta-data, at costs that scale with the number of objects rather than with object size (i.e., number of blocks in an object). Further, since the type of storage devices used by vOSD storage domains is orthogonal to the interface provided to clients, the implementation of vOSD does not require physical OSDs. The use of such new physical devices can reduce the costs of storage management in a SAN environment, since host-resident space allocation functionality of a storage domain is no longer required. On the other hand and as shown in this paper, a resource-rich realization of a vOSD storage domain can exploit the additional compute resources available at host-level to provide new and useful storage functionality to end clients and/or to shift certain computational tasks from clients to storage domains. Idle cycles typically available on storage domains [42] demonstrate the viability of this approach, as discussed further in Section 3.

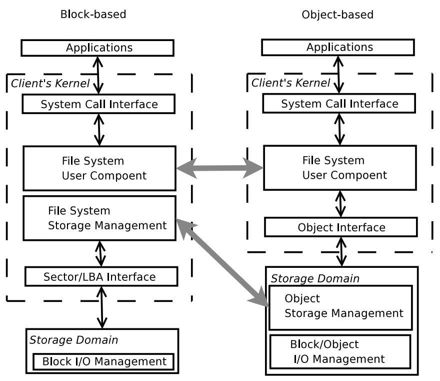

Figure 1: Comparison of block- and object-based interfaces for storage clients

2.2 Storage in Virtualized Systems

The realization of the O2S2 architecture presented in this paper is based on common methods for platform and storage virtualization. Stated explicitly, it *satisfies the storage needs of virtual machines as storage clients* by utilizing a separate storage domain onto the same physical platform as the guest. This architecture affords the common advantages ascribed to virtualized resources, including improved resource consolidation, better isolation across different applications, and reduced vulnerabilities in personal/home environments. Additional advantages include improved manageability, as argued by Chandra et al [23]. In contrast to current virtualization systems [18, 50], however, our approach uses vOSDs to replace the block-based storage interfaces, such as IDE and SCSI, currently being used. The intent, of course, is to attain the goals of improved manageability and enhanced functionality articulated earlier.

2.2.1 From Virtualized to Security-enhanced and Trusted Object Stores

In order to maintain security isolation among multiple vOSDs, the vOSD storage domain employs object-level access control. This allows sharing physical storage space among multiple vOSDs at an object granularity. This access control is Role Based Access Control (RBAC) based on *labels* or *roles*, which define the *capability* of a storage client. The storage domain maintains these labels for all the objects it stores, and utilizes an external trusted entity for the management of labels associated with a storage client. This enables the storage domain to provide access control functionality at a reduced cost.

Another key advantage of our virtualization-based realization of vOSD storage domain is that it can monitor, inspect, and manage guest VMs and their use of storage 'from the outside', using privileged domains that are not subject to the same attacks or failures faced by guests running standard operating systems and applications across open network environments. These privileged management domains or 'trust controllers' can use VM introspection (e.g., using the XenAccess [14] facility developed in our research), behavior monitoring (e.g., as done in our work on power management [37]), or I/O traffic monitoring to continually assess VM 'health' or security [40]. Such domains can even intercept I/O requests to implement new security services like firewalls or intrusion detection [9] or to re-direct certain VM actions to guarantee desired safety properties for potentially unsafe code [29].

Leveraging the abilities of access control and external monitoring provided by system virtualization, the O2S2 architecture permits storage domains to enforce desired access controls on their data. This is done by using online monitoring to establish certain 'trust' values for guest VMs and for the platforms on which they run [52], and then, enforcing access controls to ensure that data requiring certain levels of trust is accessed only by those VMs on those machines that meet those requirements. Stated differently, these 'trust' values can dynamically change the *labels* or capabilities of a client. Even though a client's label might match the label of a particular object, a dynamic label based on the original label and 'trust' value might not, resulting in declined access.

Underlying these online matching processes, of course, are basic actions taken by storage domains that (1) track (i.e., monitor) and label (i.e., compute and maintain trust-relevant metadata) the data items written and read by certain guest VMs, and (2) enforce that data items are stored and accessed only when trust values match, as per the access control or security policies stated by system administrators. A sample use case for such functionality considers doctors or nurses who create and access patient records. Here, records are labeled as per the sources that produce them, accesses require appropriate identities or roles, and in addition, they require that such accesses are only carried out from trusted platforms and guest VMs. The vOSD storage domain enforces policies like these by checking the labels (i.e., metadata) associated with data objects like patient records against the trust values of the guest VMs performing such accesses. It also enforces such properties for record storage, for instance, that certain patient records are stored on disks present at a location with better physical security.

Figure 2 shows the relationship between a distributed application running inside application VMs and a vOSD storage domain. The storage domain is a trusted entity and enforces access control itself, with the help of certain security extensions in the hypervisor. The application's behavior contributes to its "trust" value, as perceived by the trust controller and exported to the storage domain. If some part of the application runs on an untrusted platform, its "trust" value must be defined by a remote trust controller running on a trusted machine.

Fine-grained access control and metric like 'trust' are particularly relevant in data center settings, where their use extends the measures used for service level agreements (SLAs), which are typically defined to provide *statistical* guarantees on various performance characteristics of services, such as bandwidth and latency [22]. This extension is important because with multiple compute VMs [1] or storage services [2] hosted by the data center, service clients can no longer control those services' uses of data center resources. In this context, a secure and trusted vOSD storage domain can provide strong guarantees to a VM that houses sensitive information (e.g., patient records and proprietary art work) that this information will only be stored on some few *identifiable* disks, thereby reducing the risk of data being stolen or being retained after the client's run has completed. Further, when upholding the integrity of a client's actions in a virtualized environment, if data is erased by the client, the service provider must ensure that none of this data is left anywhere in the system. This can be achieved with an improved accounting by storage

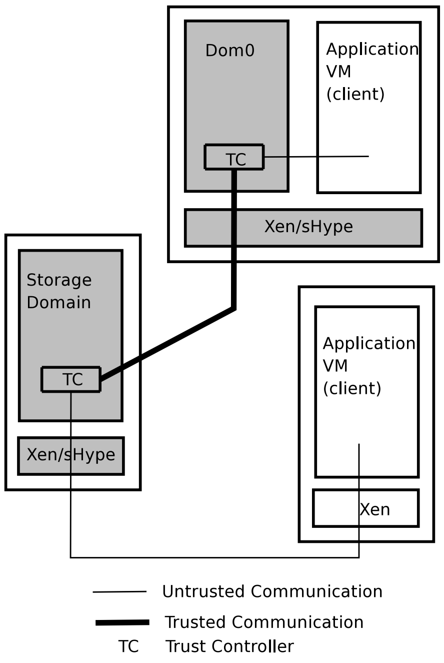

Figure 2: Storage domain as a trusted object store. Entities in gray are trusted

domains about which client's data is stored on which physical media. Such accounting functionality can also help with data recovery upon loss, media recovery, and similar tasks.

Finally, a trusted vOSD object store can be used to enhance trust management itself. In particular, a VM's access log maintained by the vOSD storage domain can be used to generate a *behavior profile* for the VM at object granularity. Such profiles are easier to manage and more scalable than those based on block-level information [40], since monitoring need not deal with client-specific information, e.g., the file-system layout. Behavior profiles can then be used to derive 'trust' values for the VM. Also, in case of a security concern, such as a world-wide virus spread, these profiles can be quickly disseminated, as signatures, to preemptively stop damaged domains from being run or used, until the problem is corrected.

An extension of the security-enhanced vOSDs presented in this paper concern the auditability of such assurances. This may require specialized hardware such as Write-Once-Read-Multiple (WORM) devices [55] and an open logging infrastructure with access provided to clients. With such support, clients can then corroborate the actions taken by the service provider in response to their own actions, and they can ensure that the identities of the physical devices used match the ones enforced by the SLA. Furthermore, immutable content on WORM devices can be upheld legally in case of disputes.

2.3 Usability in Personal/Home Environments

As stated earlier, the implementation of a vOSD does not rely on specific storage media or subsystems. This affords us with substantial advantages in environments that employ diverse storage devices and where device usage depends on dynamic measures like current context. In home or personal environments, for instance, examples include a user storing media files on a video/mp3 player, personal contact information on a cell phone/PDA, running applications from local hard disk, and archiving information on a high capacity USB hard disk and/or on DVD media.

The O2S2 architecture can easily exploit diverse devices used in dynamic settings, where based on the contextual properties of each object designated by its owner and that of storage devices, the vOSD storage domain finds the best match for storing an object in a client oblivious manner, thereby relieving application VMs from making these decisions. One method is to store objects based on their performance metrics and/or on the performance properties of storage devices, in a manner similar to that of Stonehenge [30]. Another method uses access control based on labeling, which makes it possible to consider personal devices as potential storage media, by ensuring that certain data is stored and/or accessed only on certain devices. As a concrete scenario, consider user 'A' who has connected his iPod to her home PC. The iPod is labeled with the contextual label 'media' and with the access control label 'VM A'. The virtual machine owned by 'A' is also labeled as 'VM A'. The vOSD storage domain, then, makes the content of the iPod available via the virtual disk for 'VM A'. Also, if 'A' downloads a multimedia content from the internet and sets its contextual label to 'media', the content will automatically be stored on the iPod. From then on, this content will be available to 'A' 'on the go'. In contrast, all temporary files created by 'VM A' are stored on the generic hard disk. In this manner, the vOSD storage domain performs the semantic aggregation of hard disk and iPod to present a single vOSD to 'VM A'.

In summary, the O2S2 architecture provides enhanced storage for clients in virtualized environ-

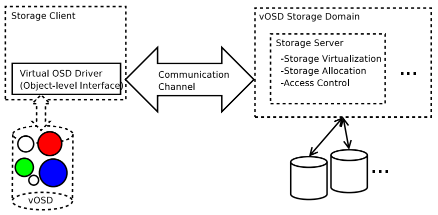

Figure 3: O2S2 architecture

ments. A key component of this architecture is its object-based interface for virtualized storage access for client VMs, providing benefits that include improved resource consolidation, better usability in heterogeneous environments, and object-level access control and trust management.

3 Architecture

The Object-Oriented Storage System (O2S2) has three main components:

- client-side virtual object-store device (vOSD) and the associated access interface provide to guest VM.
- communication channel between virtual object-store device and storage domain.
- the vOSD storage domain.

Figure 3 depicts relationship between these components.

A storage client uses a vOSD-specific interface to initiate storage requests, such as the creation/removal of objects and I/O on their content. These requests are communicated to the vOSD storage domain by the vOSD client-side driver using the communication channel. The job of the vOSD storage domain is to service these requests and provide mediated access to the physical storage devices.

The vOSD storage domain works as a *backend* for client vOSDs. It is a distributed service and is composed of one or more storage servers. Each storage server has multiple sub-components, described as follows:

- Storage virtualization. This component implements the support of multiple client vOSDs over shared physical storage and any conversions that might be required for client specific interfaces, such as converting data read from local disks into NFS read responses. This component also works as the back-end of the vOSD, in that any action performed on the vOSD is received by this component. These actions are checked for appropriate access restrictions and converted into requests for the storage management component.
- Storage management. This component facilitates the management of physical storage devices. In particular, it implements the allocation of physical storage corresponding to the objects in the vOSD and it implements any operations on them, including I/O.
- Access control Module (ACM). This component enforces per-object access control and is a key element for implementing security, privacy and trust for the clients. The basis for access control are *labels* attached to clients and to objects. These labels behave as *capabilities*. Each object contains one or more labels, which are matched according to a policy with the label of the clients accessing the object. Labels associated with clients are *not* provided by the ACM itself, and are not part of the communication protocol between the vOSD storage domain and the storage client. Rather, *clients' labels are provided by an external trusted entity, the hypervisor*. This delegation greatly simplifies the storage domain architecture, since ACM need only implement enforcement, and not deal with label (capability) management of clients at all. Issues related with this management include capability generation, dissemination, enforcing expiration, and dealing with security aspects of the communication protocol, such as spoofing and replay attacks [27]. In contrast, other storage systems, such as Lustre [6], implement this management of client capabilities as a part of the storage system itself.

  Such access controls are in addition to any access controls implemented by guest virtual machines based on certain user credentials, such as user and group identifiers.

4 Implementation

Our prototype implementation of O2S2 architecture is based on the PVFS file-system [21, 33]. It runs on a platform virtualized with Xen [43]. Figure 4 shows the different components of this implementation.

A distributed file-system is chosen as a way to implement the prototype because such file-systems are

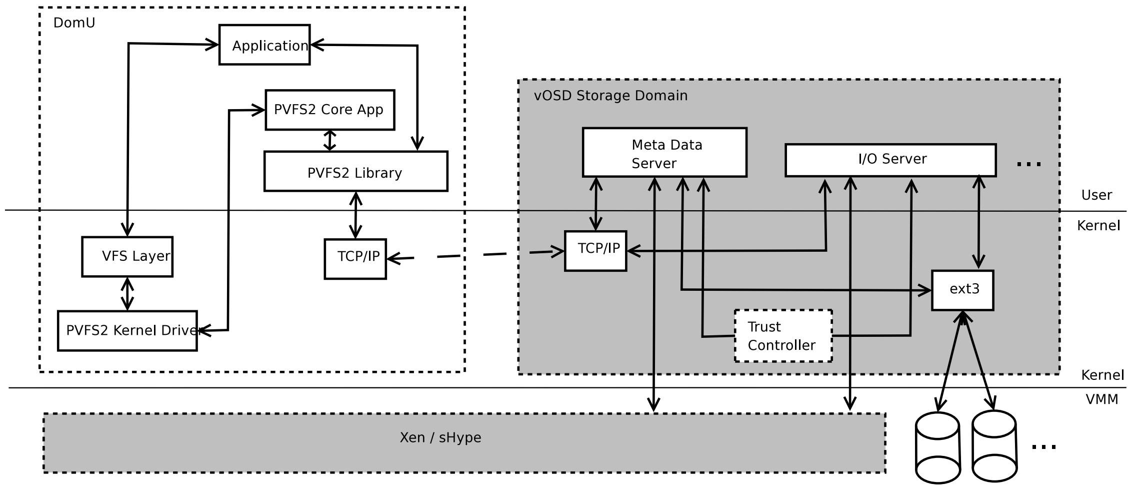

Figure 4: PVFS based object-oriented storage system

commonly used in distributed environments to implement storage solutions that (1) enable resource consolidation on servers and (2) allow data sharing among multiple clients. Such sharing is enabled by the file-system's provision of a higher-level abstraction to clients. Our choice of the PVFS cluster file-system, rather than using NFS or Coda, is due to its ability to separate meta-data and data, and because it makes it possible to distribute data among multiple servers for performance. These properties also enable extensions that can provide differentiated storage. We specifically chose PVFS2 since it is mostly implemented in user space, which makes it easy to modify and debug, unlike, e.g., Lustre [6], which is implemented in the kernel. Further, PVFS2 is a freely available, mature, and stable product.

4.1 PVFS Background

The client side vOSD is provided as a PVFS volume where objects are stored as PVFS files. The core of the PVFS file-system is implemented in user space. User-level applications can use a PVFS specific API and its client library to make PVFS system calls to access these files. These files can also be accessed by unmodified user space applications relying on the kernel's VFS layer to hide file system details. To enable this, the PVFS file-system provides a kernel driver that registers the file system with the kernel's VFS layer. This enables mounting the vOSD in the file hierarchy of the client. Any I/O in vOSD file space is directed to the PVFS kernel driver by the VFS layer. The kernel driver marshals the VFS request into a PVFS request and communicates it to a user-level application, called the *PVFS core*. This application works as a proxy to make PVFS system calls on behalf of the kernel driver. Similarly, all responses received by this application are communicated back to the driver, which converts it into appropriate VFS responses and hands it to the VFS layer.

Currently, the PVFS client component does not interface with the client kernel's page cache. Bypassing the page cache makes the file system design simple, since there is no client level consistency to maintain in a shared environment. However, this also means that every VFS request must be communicated to the storage domain. This reduces the performance for I/O operations with small block sizes. In contrast, PVFS performs well for large block sizes and large files.

The vOSD storage domain is implemented as multiple storage servers, each of which is a PVFS server. In our prototype implementation, all of these servers execute in the Dom0. Hence, the vOSD storage domain provides an example of a host-based iConnect realization [44].

A PVFS server is a user-level entity and is classified as either a meta-data server (MDS) or a I/O server. As the name suggests, a MDS manages meta-information about a PVFS file, such as attributes (length, last modification timestamp etc.) and access-control information (*label* of the file, list of user ids allowed access etc.). Some of the meta-information depends on the type of file. For example, for a regular PVFS file, its meta-data contains information about the I/O servers that are in use for storing the data. PVFS also supports extended attributes, which can be user or system defined arbitrary key-value pairs. I/O servers are used to store actual data associated with a PVFS file.

PVFS servers use a combination of Berkeley DB [39] and files in the underlying file system – the former is used to store meta-data whereas data is stored in the latter. The PVFS filesystem might create multiple chunks for a PVFS file in order to parallelize access to data, each of which is a file stored at an I/O server.

Each PVFS server manages a storage space, which is a part of the local file system that it uses to store data. In our implementation, different storage spaces reside on different physical disk partitions. These disk partitions can be spread among multiple disks.

The communication between a PVFS client, i.e., a guest VM, and PVFS servers, i.e., the storage servers of the vOSD storage domain utilizes TCP/IP networking over virtual NICs provided by Xen.

4.2 Extensions to the PVFS-based Storage Domain

The vOSD storage domain that provides enhanced object-based storage is realized as an extension of the core PVFS implementation.

First, additional attributes, called *IOHints*, and access control labels, called *acm-labels*, are associated with a PVFS server's storage space. By associating a particular physical disk partition to a storage space, these IOHints and acm-labels also extend to the level of a physical disk partition. This is the lowest granularity of meta-data managed by the vOSD storage domain.

Second, with each PVFS file object stored in client's vOSD, two extended attributes are associated – *user.iohint* and *system.acmlabel*. The former is modifiable by the user, while the latter is only modifiable by the vOSD storage domain. We also extend the PVFS client library, and correspondingly the server-side PVFS implementation, to include a file-system call, called *extended create* or *ecreate* to create a file with specified extended attributes.

The ACM is implemented by extending the PVFS server to include certain checks. These checks are included in the *prelude* part of every request serviced by the server. The type of access control implemented is Mandatory Access Control [26], where access rights of an object are non-transferable from one client to another without the intervention of the hypervisor and the vOSD storage domain. The ACM utilizes the information provided by the *secure hypervisor* (sHype) extension of Xen hypervisor [47] and, optionally, by the trust controller.

Xen's sHype extension implements a repository of one static label per VM and one static label per physical resource, such as NIC and harddisk partition. Based on a matching policy, it also enforces mandatory access control – a VM can only access a physical resource if they both have the same label. These labels can be viewed as roles, hence this type of access control is an example of Role Based Access Control (RBAC) [26]. Currently, Xen supports bind time access control, i.e., labels are only matched at the time a guest VM is created. Also, the label associated with a VM does not change for the lifetime of a VM. This may change in the future, e.g., based on the identity of the physical platform on which the VM executes, a VM's label could change.

Since sHype does not define labels for anything other than VMs and physical resources, services like the vOSD storage domain must currently implement access control themselves. Toward this end, we use sHype as a repository for guest VM labels. The vOSD storage domain keeps its own labels for each server's and for each PVFS file object stored, as described earlier. Also, since the information sHype only maintains label information for clients specific to a physical platform, it implies that all storage servers of the vOSD storage domain need to coexist on the same platform. However, it is possible to distribute them among multiple machines by incorporating a distributed trust management solution, such as shamon [35] in the O2S2 architecture.

4.2.1 Trust Controller

A trust controller is an optional component that can be utilized with the storage domain to enhance its access control enforcement. The functionality of a trust controller is to maintain a "trust" value for a guest VM. It utilizes one or more monitoring components that provide behavioral information about the guest VM. The "trust" value is a function of current behavioral information and of a trust policy. In our prototype, behavioral information is captured using a network monitoring component (*netmon*), which resides in the Dom0 kernel and essentially, extends the Xen virtual NIC backend by monitoring the network traffic to/from a guest VM.

Figure 5 shows the interaction between netmon and the trust controller. Netmon operates as fol-

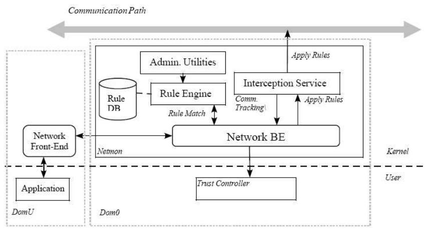

Figure 5: Interaction between trust controller and *Netmon*

lows. Based on a rule engine, it intercepts certain packets. Contents of these packets, such as headers of different protocol stacks and payload, generate the desired behavioral information. Further details about the netmon prototype along with performance analysis are described elsewhere, as part of the ProtectIT framework [32].

The current netmon prototype provides information pertaining to remote access to a guest VM, such as the number of open telnet and ssh connections, and the amounts of data transferred by these open connections. This information is exported to the trust controller using the */proc* interface. Additionally, netmon notifies the trust controller proactively when the information monitored by netmon changes. Since the trust controller is a part of the storage domain prototype in user space, kernel space netmon utilizes standard kernel-to-user space asynchronous signals for notification. The trust controller, then, accesses the network information via the /proc interface, and updates the "trust" value of the guest VM based on a specific trust model. These updates in "trust" value affect the overall access control for the guest VM, depending on the specific policy in use by the storage domain. An example trust model, along with example access control policies are described in detail in Section 5.1.

4.3 Discussion – Alternative Choices

Although our current prototype of the O2S2 architecture uses PVFS, the architecture itself is generic and can utilize alternative means to implement its components. Some of these alternatives are discussed in this section. Table 1 also summarizes these alternatives.

The client side virtual object-storage device (vOSD) is characterized by the interface it provides to the client kernel. As an alternative to the file-system API, it could utilize a device access protocol, such as T10 [56], which is an enhancement of SCSI. In this case, the device driver sets up the virtual device as part of the SCSI device stack as a SCSI initiator. A shim file system layer [54] is required to present a file-system interface on top of this virtual device, which then interfaces to the VFS layer in the kernel. Alternatively, as is the case with PVFS and other distributed file-systems, this device can also be directly accessed by user space applications via libraries.

The client vOSD driver formats storage requests as messages based on a specific protocol and forwards them to the storage domain. For example, a vOSD based on NFS can use NFS-specific messages to communicate with the storage domain. However, it is also possible for a vOSD to provide an interface to the client that is entirely different from the one used to access the storage domain. For example, a vOSD providing a PVFS interface to the storage client could convert PVFS-specific messages to T10 commands that can then be sent to a storage domain which works as the OSD target [24]. Another example of such a protocol conversion is when the storage domain performs this conversion prior to performing actual storage virtualization, management and access control tasks [28].

Since the vOSD storage domain is a distributed service, there are many ways to implement each storage server: as a user-level application (e.g., our current prototype), as a kernel-level service (e.g., Lustre), or as a runtime inside a dedicated and specialized VM of its own. Many commercial SAN and NAS implementations also fall into the first two categories. Although we are not aware of an example of a storage domain comprised of multiple specialized VMs, an approach similar to that of Libra [17] is also feasible for implementing a storage server. IBM's Tiburon project [12] uses a similar approach.

Another related issue for implementing the vOSD storage domain concerns the malleability of the enhanced functionalities being exported (i.e., whether there is a predefined set and/or whether a client must choose among them, or whether the set can be defined by the client). Since these functionalities are domain specific, e.g., they may differ for multimedia vs. file storage, there are multiple approaches for providing an interface for defining these functionalities. Examples include a domain specific language [3] and arbitrary binaries executing inside isolated containers, such as processes and VMs. In this work, we focus on a fixed set of functionalities offered by the vOSD storage domain – the dynamic extension of storage domains by a client is part of our future work.

Table 1: Summary of design choices for various components of O2S2 architecture

Component	Choices
vOSD interface to client	Filesystem (NFS, PVFS), T10
Communication Protocol between the vOSD storage domain and vOSD	Filesystem specific, T10 based
Storage Server	User process, Kernel service, Specialized VM
Objects for storage management	Files on local file-system, OSD objects

The job of the storage virtualization component is to map a vOSD object access request to a set of objects managed by the storage management component. The storage management component manages these objects with the help of the underlying platform's storage services. For example, our current PVFS based prototype and many other cluster file-system servers use local file systems, such as ext3, of the machines on which they run to store these objects as files. If the underlying platform has OSDs attached to it, these objects could be provided by the device itself. In this case, the storage management component runs on the OSD that houses the particular object being accessed. In a similar fashion, ACM could be located on the device itself.

5 Functionalities

This section describes various functionalities of the vOSD storage domain implementation and demonstrates how these functionalities provide enhanced vOSDs to storage clients.

5.1 Object-based Access Control

The vOSD storage domain implements per-object, multi-layer, role based access control (RBAC). RBAC is based on *labels* associated with clients, physical storage devices, and individual objects stored in those devices.

On each request, a storage server first determines the client's label. Currently, each client is a VM physically located on the local machine with IPs in a private subnet. By using the association of a client's IP address with its VM id, the storage server obtains the client's label from Xen/sHype. This label is then cached for the lifetime of the VM. Next, this label is matched against labels of the storage space being managed by the server. In case of a mismatch, the request is denied. Otherwise, if a request does not pertain to a specific object in the storage space, such as a request to obtain the PVFS configuration from the storage domain's MDS, the access control returns success, and the request is allowed to continue.

If a request pertains to a specific object, labels of that object (stored as system.acmlabel extended attributes) are matched with the client's label. In case of a mismatch, the request is denied, otherwise is allowed to continue.

For each PVFS file created by the client in vOSD, all of the objects stored in the storage space of a server inherit the client's label. For example, for a file create operation by a client with label *WebSurfing*, the meta-data object and one or more data-objects, created on MDS and I/O servers respectively, contain system.acmlabel attribute set as *WebSurfing*. It is possible to append more labels to an object from a privileged client (such as Dom0) – hence an object can be shared among multiple clients. Alternatively, a "group" type label can be utilized for sharing purpose, where the hypervisor maintains the association of a client to a group, and the individual object is labeled with the "group" label. The latter approach is currently part of our future work.

A client's label need not be statically defined. It can be dynamically computed based on a function of the initial static label assigned by the VMM and the "trust" value of the client, as determined by the trust controller. This dynamic label is equivalent to a dynamic role assignment for the client – hence the access control implemented by the storage domain based on dynamic role assignment can be viewed as a special form of RBAC, termed dynamic RBAC. Dynamic RBAC provides more flexibility than static RBAC, which only utilizes static labels.

The trust controller computes the "trust" values of the guest VM based on a simple trust model, which defines floating values for "trust" in the range $(0.0, 1.0]$. These "trust" values are based on the number of open telnet and ssh connections with the VM as the server, computed according to the following formula:

$$\text{"trust"} = 1/(1+\text{number of active ssh connections}+\text{number of active telnet connections}) \quad (1)$$

The information related to the number of open connections is provided by netmon, as described earlier.

Based on this floating "trust" value and the client VM's static label (label_{static}), the storage domain can use different policies to generate client VM's dynamic label ($\text{label}_{dynamic}$). Two such policies and their application scenarios are described next.

Policy 1

```
if "trust" == 1.0 then
    $\text{label}_{dynamic} = \text{label}_{static}$
else if "trust" < 1.0 then
    $\text{label}_{dynamic} = \text{NULL}$
end if
```

A NULL label by default denies any access. Policy 1 implies that when a VM has any open ssh or telnet connections, it cannot access any object in the vOSD.

Policy 2 defines dynamic labels for different "trust" values. A dynamic label is constructed by appending access restrictions to the static label. Access restrictions are represented as an AND of NOTs $((!T_1)\&(!T_2)\&\cdots\&(!T_n))$, where each $T_i, i \in 1, n$ is a specific access type. Examples of these access types used in policy 2 are:

- DW – data write,
- DR – data read,
- MDRW – meta-data read write.

For example, a dynamic label $\text{label}_{dynamic} = \text{label}_{static}$ *& (!DW) & (!DR)* evaluates to NULL for access types of data write and data read, but evaluates to label_{static} for meta-data read write. This dynamically evaluated label is then matched against the object's label for access control purposes.

Object-based access control enables the efficient sharing of physical devices, since the granularity of access control can be per-object rather than targeting larger disk partitions. Hence, objects from multiple vOSDs can be stored on the same physical disk, which reduces fragmentation and increases utilization. Second, objects from multiple vOSDs can be shared among multiple clients – by assigning multiple labels to this object pertaining to multiple clients.

Policy 2

```
if "trust" == 1.0 then
    $\text{label}_{dynamic} = \text{label}_{static}$
else if "trust" $\geq 0.50$ then
    $\text{label}_{dynamic} = \text{label}_{static}$ & (!DW)
else if "trust" $\geq 0.33$ then
    $\text{label}_{dynamic} = \text{label}_{static}$ & (!DW) & (!DR)
else if "trust" $\geq 0.25$ then
    $\text{label}_{dynamic} = \text{label}_{static}$ & (!DW) & (!DR) & (!MDRW)
end if
```

Third, a storage domain level access control can be a part of a multi-layer access control solution [41]. For example, a client may impose further role-based access control for multiple users, such as in SELinux [8].

5.2 Semantic Aggregation of Multiple Storage Devices

In a heterogeneous storage environment, e.g., a home/personal environment, the vOSD storage domain collectively utilizes the ensemble to storage devices to store objects from various vOSDs. An object is stored on the storage space of a specific storage server, based on its attributes (e.g., *user.iohint* and type), and *IOHints* of the storage server. For example, an object with user.iohint attribute set to 'X' can only be stored at server(s) whose storage space contains 'X' (or *generic*) as a IOHint. These labels can be chosen based on any policy, e.g., physical device specific characteristics. Such semantic aggregation of multiple storage devices based on objects' properties enables *differentiated storage* for a vOSD. Here, costs associated with deciding which storage server to use are one time and are paid at the time when an object is created. Afterwards, an object's I/O performance depends on the I/O performance of the specific storage server(s), which itself depends on the share of resources (such as disk bandwidth and CPU) associated with these server(s).

Differentiated storage provides multiple benefits, including easier data management and performance isolation among multiple clients and multiple types of applications in a client. Performance isolation is provided by storing isolated objects on different servers using different disks for storage space. An example is storing metadata on a MDS using a faster flash based disk, while storing sequential data on an I/O server using a high capacity SCSI disk. In this fashion, the vOSD storage domain can minimize the performance impact of extensive meta-data I/O performed by a VM, e.g., searching for a particular file, on a VM that performs streaming data I/O, e.g., watching a movie. Easier management results from the fact that data can be stored on a device based on its utility, as suggested earlier in Section 2.

6 Experimental Evaluation

Experimental evaluation of the prototype O2S2 implementation is carried out on a dual-core, 3GHz, 64-bit x86 CPU based server class machine with 1GB RAM and 160GB SATA, 7200 RPM hard disk with 8MB cache. The hypervisor used is Xen version 3.0.4 with sHype enabled. The policy used by sHype is a simple type enforcement policy. The privileged VM, Dom0, is assigned 512MB RAM and exclusive access to one physical processor. The second processor is shared among different guest VMs. For our experiments, each guest VM is configured with 128MB RAM. Both Dom0 and guest VMs run a para-virtualized Linux kernel based on version 2.6.16.33. The vOSD storage domain, which resides in Dom0's user space, and vOSDs it provides to a client are based on PVFS version 2.6.3.

6.1 Microbenchmarks

These experiments measure the basic costs of implementing enhanced functionality in the storage domain, namely object-level access control and differentiated storage. For the latter, we also provide a quantitative evaluation of the potential benefits in terms of the performance isolation it provides.

6.1.1 Access Control Module

Performance For analyzing the costs associated with ACM, we execute various PVFS system calls on a vOSD from a guest VM, with and without ACM present in the vOSD storage domain. The storage domain runs two storage servers – one MDS and one I/O server, both sharing the same disk for their storage space. The application executing these system calls directly uses PVFS's client libraries, without having to go through the kernel's VFS layer.

Figure 6 depicts the normalized latency comparison for some of these system calls with and without ACM, using the latter as the base. The figure also includes the total number of access control checks performed for each system call, followed by its name on the x-axis in the following format: (#Server ACM checks+ #Object ACM checks). Since ACM caches a client's labels from Xen/sHype, the cost of a Server ACM check involves label matching only. The cost of an Object ACM check requires accessing its extended attribute (*system.acmlabel*) from PVFS's storage, followed by label matching. We find the cost of a Server ACM check and an Object ACM check to be $\sim 8\mu s$ and $\sim 62\mu s$, respectively. Also, the latency of read (write) system call depend on the block size being read (written). Since the cost of ACM checks is fixed per system call, the normalized latency with ACM checks for read and write system calls will change based on the block size. The measurements presented in Figure 6 are the costs for 32MB size blocks. Since the application makes single blocking I/O requests, I/O throughput can be computed as (block size/latency of system call). For this block size, we find the read and write throughput to be 170.5 MB/s and 40.81 MB/s, respectively, with ACM, as compared to 174.81 MB/s and 40.98 MB/s, respectively, without ACM. These results demonstrate that the ACM component minimally impacts the performance of PVFS system calls, both in terms of latency and throughput.

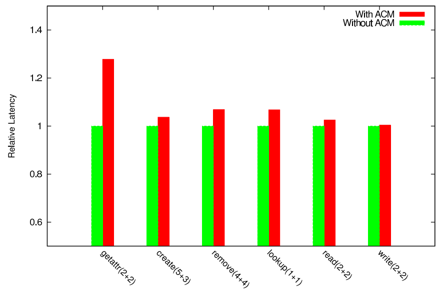

Figure 6: Performance comparison of PVFS system calls with- and without-ACM in the vOSD storage domain

Scalability We use write throughput to demonstrate the impact of ACM component on a storage domain's scalability. Figure 7 shows the relative performance profile of a storage domain with increasing number of VMs, using single VM measurements as the base case. Based on memory size, the maximum number of VMs configured with 128MB RAM handled by our test platform is 3 (theoretically, the limit is 4, given that there is no memory overcommitment; however, the VMM uses a certain amount of mem-

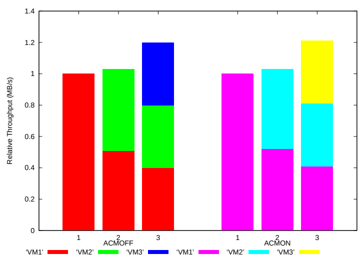

Figure 7: Scalability of the vOSD storage domain.

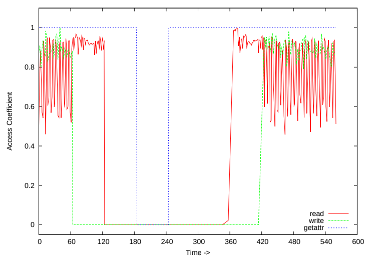

Figure 8: Effect of dynamic RBAC on different workloads

ory itself, resulting in a limit of 3). Here, single VM numbers for both, with- and without-ACM, cases are normalized to one. In case of more than one VM with ACM (without ACM), cumulative performance of all VMs relative to the single VM with ACM (without ACM) is shown, along with individual components. A near-identical performance profile demonstrates that adding ACM functionality does not impact the scalability of a storage domain.

Dynamic Access Control As stated earlier, the ACM module offers dynamic role-based access control (RBAC) functionality. This is demonstrated by dynamically changing the network related behavior of a guest VM and by showing its effect on three storage workloads running in the guest VM. The first workload continuously writes to a file; the second workload continuously reads from a file; and the third workload continuously reads attributes of a file (i.e., performs meta-data reads). These workloads are identified as *write*, *read* and *getattr*, respectively. Figure 8 shows the time line on X-axis and behavior of workloads on the Y-axis in terms of *Access Coefficient*. A value of access coefficient greater than 0.0 indicates successful access, while a value of 0.0 indicates an access failure. For data read and write workloads, the access coefficient is computed as the ratio of instantaneous throughput and maximum throughput achieved. For meta-data read write workload, success of access results in access coefficient value of 1.0, while failure of access, in 0.0.

A script executing in the Dom0 incrementally opens four ssh connections to the guest VM, and then incrementally drops them. The script opens connections one and two at 60 seconds and 120 seconds, respectively, and opens connections three and four at 180 seconds. These connections are dropped at 60 seconds intervals starting at 240 seconds. These dynamic changes in number of network connections affect the "trust" value of the guest VM, computed based on the Equation 1. The storage domain uses policy 2 to define dynamic roles for the guest VM, which affects its access to the objects stored in the vOSD.

As demonstrated by the results, the data write workload can successfully function expect in time period $(60, 420)$ seconds. This is due to the fact that the "trust" value of the guest VM in this time period remains < 1.0, which prohibits write access to the objects for the VM. Similarly, the data read workload and meta-data read workload can successfully function expect in time periods $(120, 360)$ and $(180, 240)$, in which the "trust" values are < 0.50 and < 0.25, respectively. These results show the ability of the storage domain to dynamically adapt the access control imposed on a guest VM with changes in its "trust" value, as perceived by the trust controller.

6.1.2 Differentiated Storage

For differentiated storage, we compare the latency of two PVFS system calls – *ecreate* and pre-existing *create*, both performed from a user-level application inside a VM, directly using the PVFS libraries. In this experiment the vOSD storage domain consists of three storage servers – one MDS, one I/O server with label *mobile*, and other I/O server with label *fixed*. Based on the value of the *user.iohint* attribute specified with the *ecreate* call, the storage domain chooses one of the I/O servers for storing the object data. With the *create* call, the client chooses an I/O server in a round-robin fashion. Table 2 shows the latency of these system calls, as measured from the client VM. The difference between the latencies is due to extra processing performed in the MDS to match the attribute with *IOHints* of various I/O servers, and due to additional I/O in the MDS to store the extended attribute, the latency of which is based on the disk being used for the storage space at the MDS. For this

Table 2: Cost of implementing differentiated storage

Operation	Latency (ms)
create	19.504
ecreate	22.393

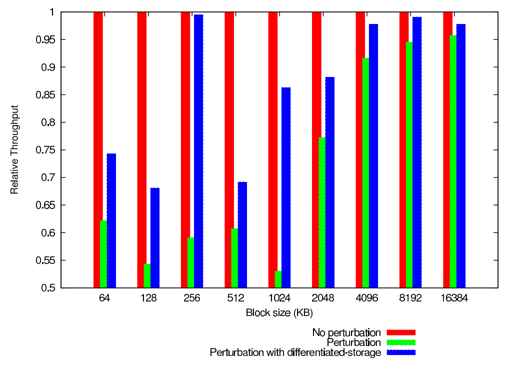

Figure 9: Performance isolation in vOSD storage domain

experiment, all storage servers share the same SCSI disk. We expect the latency to be lower for different kinds of storage, such as a flash disk, which provides better performance for short, random, reads of extended attributes [51].

In order to show the performance benefits of differentiated storage, we run the vOSD storage domain with two servers, one MDS and one I/O server. There are two competing VMs performing different kinds of I/O activities. One VM, VM1, is continuously creating new extended attributes for an object, while the other VM, VM2, is doing writes to the same object. We plot the performance of VM2 executing simultaneously with VM1 in two scenarios – one, where both MDS and I/O server share the same SCSI disk for their storage space, termed 'perturbation', and two, where MDS uses a ramdisk for its storage space while the I/O server uses the SCSI disk, termed 'perturbation with differentiated-storage'. The throughput of VM2 without the presence of VM1 is used as the base line, labeled as 'no perturbation', and the performance of VM2 in scenarios described above is plotted relative to this base line performance. Figure 9 shows VM2's write throughput for different block sizes.

These results demonstrate that VM1 can substantially degrade VM2's I/O performance, since both MDS and I/O server must share the disk I/O path, along with the CPU. However, with differentiated storage, the vOSD storage domain provides much better performance isolation. Performance impact

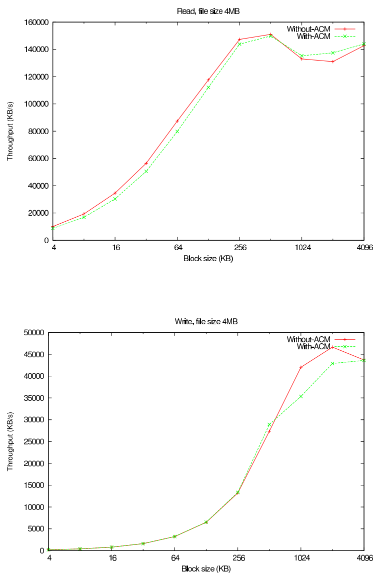

Figure 10: IOzone small I/O performance

with differentiated storage emanates from the fact that the MDS and the I/O server share the same CPU to serve VM1 and VM2, respectively. We expect the performance with differentiated storage to be even closer to the base line in future multicore machines, where different storage servers can be provided with separate physical CPUs.

6.2 IOzone Benchmark

To understand the performance implications of the ACM component on application-level workloads, we evaluate the I/O performance of our prototype storage domain using the IOzone benchmark [5]. The IOzone benchmark measures file I/O performance for many file-system operations, such as read, write, and mmaped I/O. This benchmark is executed on a client VM, and accesses the vOSD via kernel's VFS interface. The goal of these experiments is to demonstrate that (1) using ACM component does not significantly impact the I/O performance of the vOSD storage domain, and (2) the vOSD storage domain can provide significant performance benefits to a client VM, along with enhanced functionality, by utilizing Dom0's computational resources, which is an option that is not available to a block-based virtual disk (VBD) based on a physical disk partition.

For IOzone's write throughput experiments, we also enable an additional parameter for storage servers, called *NoDataSync* (NDS). This option allows storage servers to buffer writes before they are flushed to the disk. Without this option, every write is flushed to the disk. Using this option allows the vOSD storage domain to provide much better performance by reducing the latency of each write operation. The tradeoff is that the storage domain may loose data in the event of a server failover. However, using redundancy with this option enabled could provide similar performance benefits, at a reduced risk of failure.

As mentioned earlier, the PVFS file system does not utilize a client's page cache, and hence the I/O throughput of the vOSD storage domain for small block sizes is limited – anywhere from .8% upto 14% of that of a VBD. Keeping this limitation in mind, for small read and writes, we only demonstrate the comparative performance of the vOSD storage domain with- and without-ACM. Figure 10 depicts read and write throughput with varying block sizes for a file of size 4MB. Since ACM imposes a fixed cost on each I/O operation, with increasing block sizes, overall cost for access checks decreases for a fixed file size due to a decrease in total number of I/O operations being performed. The *relative performance*, measured as small quantity/larger quantity for each block size, varies within $(86.5\%, 99.3)$ and $(84\%, 100\%)$ of each other, for read and write respectively. These results demonstrate that the ACM component minimally impacts the I/O performance of the vOSD storage domain.

For large I/O, we include results for two file sizes, 128MB and 512MB, respectively. We also include the results for VBD. These results are shown in Figures 11 and 12. For a 128MB file, the client VM's page cache is no longer effective. Hence the performance of read for VBD is around 45MB/s. However, vOSD continues to enjoy the benefits of the vOSD storage domain's page cache, and hence can provide read throughput of upto 150MB/s. For writes, the performance of vOSD without NDS trails the performance of VBD, since every write must go to disk in both cases – however, vOSD's path to physical disk is longer than VBD. However, vOSD with NDS uses asynchronous write in the storage domain, and hence can provide write throughput of $\sim$ 120MB/s, upto $\sim 2X$ of that of VBD.

For a 512MB file, both the client VM's and the vOSD storage domain's page caches are no longer effective, but larger memory in the storage domain still enables better performance for vOSD for large block

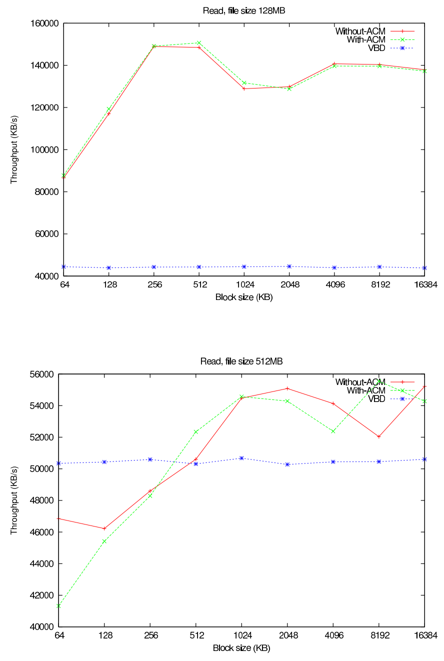

Figure 11: IOzone large read performance

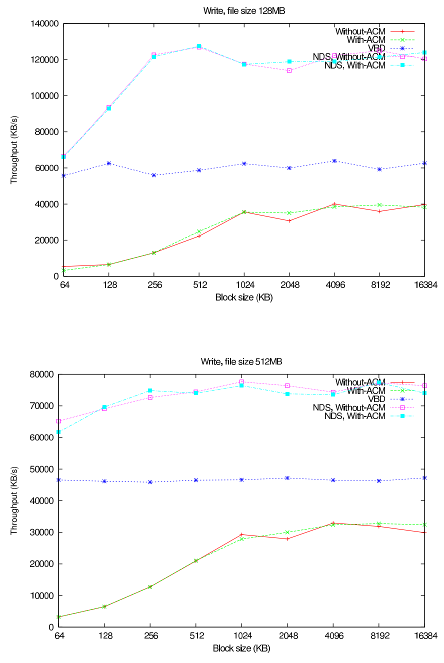

Figure 12: IOzone large write performance

sizes, since virtual network I/O is much faster than hard disk access. For writes, vOSD without NDS is slower than VBD, for the same reason as described above. However, vOSD with NDS can overlap asynchronous writes with disk I/O, and hence can provide $\sim 1.5X$ performance gain over VBD.

Similar to the previous case of small file size, there is minimal performance impact of ACM component on the vOSD storage domain for large file sizes. The relative performance of read varies within $(98.5\%, 99.6\%)$ for a 128MB file, and within $(88.1\%, 99.9\%)$ for a 512MB file. Similarly, for writes, it varies within $(89\%, 99.9\%)$ for a 128MB file, and within $(92\%, 100\%)$ for a 512MB file.

The IOzone benchmark results demonstrate that the enhanced access control functionalities of the storage domain can be implemented with minimal performance overhead for the guest VMs. In particular, the throughput for read and write operations degrades only minimally when utilizing the ACM component. Also, an object based storage domain makes it possible to better exploit the resources available at the storage domain as compared to a block based storage virtualization solution.

7 Related Work

The O2S2 architecture shares its object based design with many distributed file-systems, such as Lustre [6], Panasas [59] and Ceph [57]. While the main focus of these file-systems has been high performance and scalability, it is possible to extend them with enhanced per-object functionality and properties explored as part of this work. A similar approach is taken by Piernas et al [42], where they extend Lustre with active storage functionality.

Storage systems utilize various data properties as hints for storage management, such as data encoding, fault model, timing model [15] and frequency of access [31]. These properties can also influence other properties. For example, frequency of access can indicate whether certain data should be stored compressed [20], or decide reliability guarantees provided to it [60]. These properties can be similarly incorporated as hints in the O2S2 architecture at an object level.

Previous research in security management for network attached storage, both at an object-level [27] and block-level [16], utilizes un-forgeable cryptographic capabilities issues by storage servers to enforce access control. In contrast, we use a multi-layer approach, where capabilities external to the storage system, *labels* provided by the VMM, are utilized to enforce access control. These capabilities are used in a manner that is oblivious to storage clients. Our approach is similar to using external hardware components, such as Trusted Platform Modules (TPM) [13], to store and provide capabilities about an potentially untrusted execution entity.

Our work is similar in spirit to semantic virtualization of multimedia devices [45], where semantic information based on a higher level API (v4l) is used to provide sharing of content and functionality for multimedia devices. Further, service domains implementing the virtualization service (such as a storage domain, a driver domain providing network virtualization in a virtualized environment [43], or a VMedia runtime [45] for multimedia device virtualization) can use their computational resources to implement additional functionalities/properties for virtualized devices. This is not necessarily the case when these are directly supported by the underlying physical hardware itself. Examples include image manipulation in VMedia, additional computations on data in active storage, either in storage servers [42], or in the hardware itself [46], and TCP segment offload in virtual NICs [36].

8 Conclusions and Future Work

This paper presents an object-based storage system architecture, and a PVFS file-system based prototype implementation, called the *vOSD storage domain*. The storage virtualization service, enabled by the vOSD storage domain, provides virtual object-storage devices (vOSDs) to VMs in a virtualized environment. An object-based interface not only allows for *efficient sharing of physical devices*, it also enables *dynamic, role-based, access control* and *usability based performance isolation* in a heterogeneous storage environment. Performance results demonstrate that our storage domain implementation provides enhanced functionalities without adversely affecting the performance and scalability of the storage service. Further, by efficiently utilizing Dom0's resources, the storage domain can also provide certain performance benefits to client VMs, such as the use of its page cache as a storage cache.

In the future, the PVFS based interface for the client can be replaced with a standardized T10 interface [56]. This permits the vOSD to be entirely decoupled from the specific vOSD storage domain implementation. Translation between the T10 interface and the underlying storage domain's storage access mechanisms will be implemented in the vOSD storage domain itself. To this effect, Xen's virtual SCSI frontend [50] can be enhanced to make it compatible with T10, and similarly, the virtual SCSI backend can be merged with the storage domain. Additionally, it is possible to utilize a different file-system backend, LWFS [38], which promises efficient I/O in large scale systems and more flexibility for implementing per-object properties, such as checkpointing.

As part of future work, the current vOSD storage domain implementation can be extended to make it distributed, such that various storage servers can be located on different physical machines. This will require integration with a distributed trust management infrastructure, such as shamon [35]. Also, extending the vOSD storage domain with logging infrastructure for SLA auditing will enhance its security and trust related properties, as discussed in Section 2.2.1. As demonstrated by the object-based storage virtualization service, there are multiple benefits of integrating security and trust functionalities provided by the underlying platform. In order to attain these benefits for virtualization services in a distributed environment comprising of multiple physical machines, it is imperative that these security and trust management solutions themselves be distributed. Ongoing and future work in this area will address the mechanisms and policies for trust management in a distributed environment [35]. By integrating virtualization services with distributed, virtualization aware, trust management solutions, we can provide secure virtualization services for the enterprise. Another aspect of this integration involves building better trust models that accurately reflect the "trust" properties of a VM. These models will utilize the behavioral information provided by various monitoring components, such as XenAccess [40] and netmon [32]. These trust models will enable virtualization services to implement better and more meaningful dynamic access control policies.

References

[1] Amazon Elastic Compute Cloud. aws.amazon.com/ec2, accessed May, 2007.

[2] Amazon Simple Storage Service. aws.amazon.com/s3, accessed May, 2007.

[3] EVPath. http://www.cc.gatech.edu/systems/projects/EVPath/, aceessed April, 2007.

[4] First Look: Maxtor's NAS Offers Simple Solution. PCWorld Magazine article, published March 15, 2005. http://www.pcworld.com/article/id,120063-page,1/article.html, accessed May, 2007.

[5] IOzone File System Benchmark. www.iozone.org, accessed May, 2007.

[6] Lustre: A Scalable, High-Performance File System. http://www.lustre.org/docs/whitepaper.pdf, accessed May, 2007.

[7] ObjectStone. http://www.haifa.il.ibm.com/projects/storage/objectstore/objectstone.ht%ml, accessed May, 2007.

[8] Security-Enhanced Linux. http://www.nsa.gov/selinux/, accessed May 2007.

[9] SNORT Intrusion Detection System. www.snort.org, accessed May, 2006.

[10] The Linux Logical Volume Manager. http://www.redhat.com/magazine/009jul05/features/lvm2/, accessed May, 2007.

[11] The Open Archives Initiative Protocol for Metadata Harvesting Guidelines : Provenance. http://www.openarchives.org/OAI/2.0/guidelines-provenance.htm, accessed October, 2007.

[12] Tiburon. http://www.almaden.ibm.com/storagesystems/projects/tiburon/, accessed May, 2007.

[13] Trusted Platform Module (TPM) Specification. https://www.trustedcomputinggroup.org/specs/TPM, accessed February, 2007.

[14] XenAccess Library. http://xenaccess.sourceforge.net/, accessed May, 2007.

[15] M. Abd-El-Malek, I. William V. Courtright, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile Cluster-Based Storage. In *Proc. of FAST*, 2005.

[16] M. K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. A. Thekkath. Block-Level Security for Network-Attached Disks. In *Proc. of FAST*, 2003.

[17] G. Ammons, J. Appavoo, M. Butrico, D. D. Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. V. Hensbergen, and R. W. Wisniewski. Libra: a Library Operating System for a JVM in a Virtualized Execution Environment. In *Proc. of VEE*, 2007.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. of SOSP*, 2003.

[19] P. J. Braam. The Coda Distributed File System. *Linux Journal*, 50, June 1998.

[20] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-Line Data Compression in a Log-Structured File System. In *Proc. of ASPLOS*, 1992.

[21] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proc. of the 4th Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[22] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance Virtualization for Large-Scale Storage Systems. In *Proc. of the Symposium on Reliable Distributed Systems (SRDS)*, 2003.

[23] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The Collective: A Cache-Based System Management Architecture. In *Proc. of NSDI*, 2005.

[24] A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan. Integrating Parallel File Systems with Object-based Storage Devices. In *Proc. of Supercomputing*, 2007.

[25] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran. Object Storage: The Future Building Block for Storage Systems. In *Proc. of the Second International IEEE Symposium on Emergence of Globally Distributed Data*, 2005.

[26] D. Ferraiolo and R. Kuhn. Role-Based Access Control. In *Proc. of 15th NIST-NCSC National Computer Security Conference*, 1992.

[27] G. A. Gibson, D. P. Nagle, K. Amiri, F. W. Chang, E. Feinberg, H. G. C. Lee, B. Ozceri, E. Riedel, and D. Rochberg. A Case for Network-Attached Secure Disks. Technical Report CMU-CS-96-142, CS, Carnegie Mellon University, 1996.

[28] D. Hildebrand and P. Honeyman. Direct-pNFS: Scalable, Transparent, and Versatile Access to Parallel File Systems. In *Proc. of HPDC*, 2007.

[29] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-Based Protection Using Demand Emulation. In *Proc. of the European Conference on Computer Systems (EuroSys)*, 2006.

[30] L. Huang, G. Peng, and T. cker Chiueh. Multi-Dimensional Storage Virtualization. In *Proc. of SIGMETRICS*, 2004.

[31] J. T. Kohl, C. Staelin, and M. Stonebraker. HighLight: Using a Log-Structured File System for Tertiary Storage Management. In *Proc. of the USENIX Winter 1993 Technical Conference*, 1993.

[32] J. Kong, K. Schwan, M. Lee, and M. Ahamed. ProtectIT: Trusted Distributed Services Operating on Sensitive Data. Under review for publication in EuroSys 2008.

[33] R. Latham, N. Miller, R. Ross, and P. Carns. A Next Generation Parallel File System for Linux Clusters. *LinuxWorld*, pages 56–59, January 2004.

[34] X. Ma and A. N. Reddy. MVSS: An Active Storage Architecture. *IEEE Transactions On Parallel And Distributed Systems*, 14(9), September 2003.

[35] J. McCune, S. Berger, R. Caceres, T. Jaeger, and R. Sailer. Shamon: A Systems Approach to Distributed Trust. In *Proc. of the Annual Computer Security Applications Conference*, 2006.

[36] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing Network Virtualization in Xen. In *Proc. of USENIX ATC*, 2006.

[37] R. Nathuji and K. Schwan. VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems. In *Proc. of SOSP*, 2007.

[38] R. A. Oldfield, A. B. Maccabe, S. Arunagiri, T. Kordenbrock, R. Riesen, L. Ward, and P. Widener. Lightweight I/O for Scientific Applications. In *Proc. of Cluster Computing*, 2006.

[39] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proc. of USENIX ATC*, 1999.

[40] B. D. Payne, M. Carbone, and W. Lee. Secure and Flexible Monitoring of Virtual Machines. In *Proc. of the Annual Computer Security Applications Conference*, 2007.

[41] B. D. Payne, R. Sailer, R. Caceres, R. Perez, and W. Lee. A Layered Approach to Simplified Access Control in Virtualized Systems. *ACM SIGOPS Operating Systems Review*, 41(4), July 2007.

[42] J. Piernas, J. Nieplocha, and E. J. Felix. Evaluation of Active Storage Strategies for the Lustre Parallel File System. In *Proc. of Supercomputing*, 2007.

[43] I. Pratt, K. Fraser, S. Hand, C. Limpack, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick. Xen 3.0 and the Art of Virtualization. In *Proc. of the Ottawa Linux Symposium*, 2005.

[44] H. Raj, S. Kumar, B. Seshasayee, R. Niranjan, A. Gavrilovska, and K. Schwan. Enabling Semantic Communications for Virtual Machines via iConnect. In *Proc. of VTDC, help in conjunction with SC*, 2007.

[45] H. Raj, B. Seshasayee, and K. Schwan. VMedia: Enhanced Multimedia Services in Virtualized Systems. In *Proc. of MMCN*, 2007.

[46] E. Riedel and G. Gibson. Active Disks - Remote Execution for Network-Attached Storage. Technical Report CMU-CS-97-198, CS, Carnegie Mellon University, 1997. `www.pdl.cmu.edu/PDL-FTP/NASD/CMU-CS-97-198.pdf`, accessed October, 2007.

[47] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-based Security Architecture for the Xen Opensource Hypervisor. In *Proc. of the Annual Computer Security Applications Conference*, 2005.

[48] L. Singleton, R. Nathuji, and K. Schwan. Flash on Disk for Low-power Multimedia Computing. In *Proc. of MMCN*, 2007.

[49] A. Subbiah and D. Blough. An Approach for Fault Tolerant and Secure Data Storage in Collaborative Work Environments. In *Proc. of the Workshop on Storage Security and Survivability*, 2005. `http://users.ece.gatech.edu/~dblough/research/papers/storagess05.pdf`, accessed May, 2007.

[50] F. Tomonori. Xen scsifront/back Drivers. Presented at Fall Xen summit 2006, available from `http://xen.xensource.com/files/summit_3/xen-scsi-slides.pdf`, accessed May, 2007.

[51] W. Tuma. Comparison of Drive Technologies for High-Transaction Databases. `www.storagesearch.com/soliddata-art2-comparisons.pdf`, accessed May, 2007.

[52] R. Viswanath, M. Ahamed, and K. Schwan. Harnessing Non-dedicated Wide-area Clusters for On-demand Computing. In *Proc. of the IEEE International Conference on Cluster Computing*, 2005.

[53] G. Wallace, O. J. Anshus, P. Bi, H. Chen, Y. Chen, D. Clark, P. Cook, A. Finkelstein, T. Funkhouser, A. Gupta, M. Hibbs, K. Li, Z. Liu, R. Samanta, R. Sukthankar, and O. Troyanskaya. Tools and Applications for Large-Scale Display Walls. *IEEE Computer Graphics and Applications*, 25(4), July 2005.

[54] F. Wang, S. A. Brandt, E. L. Miller, and D. D. Long. OBFS: A File System for Object-Based Storage Devices. In *Proc. of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2004)*, 2004.

[55] Y. Wang and Y. Zheng. Fast and Secure Magnetic WORM Storage Systems. In *SISW '03: Proc. of the Second IEEE International Security in Storage Workshop*, 2003.

[56] R. O. Weber. Information technology - SCSI Object-Based Storage Device Commands (OSD). `http://t10.org/ftp/t10/drafts/osd/osd-r10.pdf`, accessed May, 2007.

[57] S. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proc. of OSDI*, 2006.

[58] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, Scalable, Decentralized placement of Replicated Data. In *Proc. of Supercomputing*, 2006.

[59] B. Welch and G. Gibson. Managing Scalability in Object Storage Systems for HPC Linux Clusters. In *Proc. of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2004.

[60] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Trans. Comput. Syst.*, 14(1), 1996.

[61] M. Wolf, Z. Cai, W. Huang, and K. Schwan. SmartPointers: Personalized Scientific Data Portals in Your Hand. In *Proc. of Supercomputing*, 2002.