Improving Distributed File System Performance in Virtual Machine Environments *

Xin Zhao Atul Prakash Brian Noble Kevin Borders
University of Michigan, 2260 Hayward Street, Ann Arbor, MI, 48109-2121, USA
{zhaoxin,aprakash,bnoble, kborders}@eecs.umich.edu

Abstract

Virtual machine (VM) systems have traditionally used virtual disks for file storage. Recently, there has been interest in using distributed file systems as a way to provide data storage to guest virtual machines, with the file server running on the same physical machine. Potential advantages include fine-grained data sharing, data protection, versioning, and backup to multiple guests from one central place. Unfortunately, distributed file systems often perform substantially worse than local file systems because of high network overheads, even when the file server is on the same physical machine.

This paper proposes two mechanisms to reduce communication overheads of inter-VM file system operations: *Inter-VM metadata sharing* and *Virtual Remote Procedure Call (VRPC)*. Inter-VM metadata sharing enables clients to directly read file attributes from the server without an RPC exchange. VRPC follows standard RPC format but uses inter-VM shared memory to exchange data between a file server and its clients, which cuts out a lot of communication overhead. We implemented these two mechanisms on the Xen virtual machine platform and adapted NFS version 3 to take advantage of these two mechanisms. For an Apache build workload, NFS with Inter-VM metadata sharing and VPRCs was only 6.3% slower than the Ext3 file system, while standard NFS was 31.2% slower. These results suggest that Inter-VM metadata sharing and VRPCs significantly reduce communication overhead between virtual machines running on the same hardware.

1 Introduction

In a virtual machine environment, the main form of storage is virtual disks. The virtual machine monitor (VMM), which manages the guest virtual machines, exports block-level virtual disks to the guest VMs. The VMM can translate virtual block access requests to physical disk commands very efficiently. Unfortunately, all file-level information is lost at this point. This makes it challenging to provide features such as file sharing among guest VMs and centralized file and version maintenance [27].

To overcome these limitations, recent work [27, 35] suggests using distributed file systems such as NFS [28, 25, 26], AFS [14, 22, 29], and LustreFS [1] to provide a centralized file system for virtual machines running on the same computer. With all of the files managed in a central location, it is possible to provide features such as selective exporting of file system branches to individual guests, fine-grained sharing, file versioning, virus checking, and selective backup. Having files in a central file system can also simplify virtual machine creation because large parts of guest OSes can be shared amongst guest VMs [35, 27].

Unfortunately, distributed file systems often perform substantially worse than local file systems. This is primarily because distributed file systems have to traverse the network stack to exchange requests and results. Virtual machine systems, such as Xen [2], use shared memory to speed up virtual network operations [12]. However, Xen's optimizations do not entirely bypass the network stack and still incur substantial overhead. For example, we found that on the Postmark benchmark [16], NFS running over the Xen's virtual network is over 1.6 times slower than an Ext3 [31] file system running on a virtual disk. For an Apache build workload, it is over 31% slower than Ext3.

This paper explores techniques for substantially reducing communication overhead between distributed file system clients and servers running in a virtual machine environment. We propose two mechanisms to enhance the performance of distributed file systems. Both of these mechanisms take advantage of the fact that file server and clients reside on the same physical host and can use shared memory to reduce communication costs without compromising VM-imposed boundaries. In this paper, we started from NFS as a representative of distributed file systems and modified it to take advantage of the two proposed mechanisms. We believe that our techniques could be applied to other distributed file systems without significant design changes. In the rest of the paper, we will refer to the modified version of NFS as VNFS

*This research was supported by National Science Foundation (USA) ITR Award:ATM 0325332

(Virtual NFS).

The first mechanism we introduce is called *inter-VM metadata sharing*. Inter-VM metadata sharing means that a VNFS client can read file attributes directly from an NFS server using shared memory. Traditionally, NFS clients exchange synchronous messages with a server to retrieve file attributes. These operations are time consuming and have a negative impact on NFS performance. With the inter-VM metadata sharing technique, an NFS server can grant clients "`read-only`" access to its memory frames that contain file attributes. An NFS client can then map the metadata pages into its address space, and then fetch the file attributes directly from local memory. In addition to the performance enhancement, any updates made to file attributes are seen immediately by the clients, which leads to better attribute coherency.

The second mechanism we use to enhance NFS performance is the *Virtual Remote Procedure Call (VRPC)*. VRPCs provide the same interfaces as standard RPCs, but significantly reduce overhead by using shared memory for all communication. Taking advantage of shared memory significantly reduces the number of times that data has to be copied and eliminates the overhead of a network protocol stack.

The concept of using shared memory to reducing communication overheads has been proposed by several researchers in the past in various contexts, e.g., [3, 24, 30]. But, applying the techniques to virtual-machine file systems poses a number of special challenges: data passed in file system operations can be large and of varying size; current virtual machine monitors provide limited, low-level support for data sharing and synchronization; file systems consistency semantics must not be weakened; and changes to the guest kernels and virtual machine monitor itself should be minimized. We provide a more detailed discussion of related work in Section 2. As far as we are aware, performance of distributed file systems within a virtual machine environment has not been analyzed in the past and this paper also helps to address that gap.

We implemented both inter-VM metadata sharing and VRPCs on Xen using the low-level memory sharing capability provided by Xen's grant tables [12]. We also adapted NFS version 3 to use VRPCs and inter-VM metadata sharing. Our current implementation adds 4300 lines of code to the NFS client, 3800 lines of code to the NFS server module, and 60 lines of code to the guest Linux kernel at the NFS server side. The inter-VM metadata sharing mechanism is applicable when the NFS server runs on a Linux guest OS.

To evaluate our implementation, we ran an Apache build workload and the Postmark benchmark on four different file system configurations:

- A local file system Ext3 [31] running on a virtual disk
- A standard NFS client and server that use Xen's virtual network
- A limited version of VNFS that only uses VRPCs
- A full version of VNFS that uses both VRPCs and inter-VM metadata sharing.

In both benchmarks, the local file system Ext3 is the fastest, as expected. For the Apache build, which simulates a workload of software development activity, VNFS was only 6.3% slower than ext3, while NFS was 31% slower. Overall, standard NFS was approximately 23% slower than VNFS. In the Postmark benchmark, which is dominated by operations on small files, NFS was approximately 83% slower than VNFS with the Postmark default file size range.

We also examine the impact of VRPCs and inter-VM metadata sharing on individual file-system operations. The results indicate that inter-VM data sharing provides the biggest savings for operations that primarily read file attributes, while VRPCs provide the greatest benefit for data block reads and writes, as well as updates to file attributes.

The remainder of this paper is organized as follows. Section 2 covers related work. Section 3 discusses NFS configuration and the Xen's grant table mechanism. Section 4 describes the design of the inter-VM metadata sharing mechanism. Section 5 discusses the design of the VRPC mechanism. Section 6 presents the results of our evaluation. Finally, Section 7 concludes and talks about the direction of future work.

2 Related Work

2.1 Communication Optimization

Some NFS variants like DAFS [11, 18, 19] and NFS-over-RDMA [8] use the Remote Direct Memory Access (RDMA) capability provided by specialized network interfaces in clustered systems. With RDMA support, these file systems are able to directly transfer bulk data into user applications' memory, which significantly reduces data copying overhead. In addition to using RDMA to avoid data copying, it is also possible to offload some protocol processing tasks, such as checksum computation or packet header stripping, to network hardware [17, 19], which can help reduce CPU load. This would require sophisticated network hardware, which limits wide deployment. Moreover, in a virtual machine environment, network interfaces are virtualized in software. Even if specialized network hardware is available, protocol process-

ing and data copying still cannot be offloaded to virtual network device, unless virtual network driver is extended to virtualize the special hardware features.

IO-Lite [24] unifies all cache and buffer pages and allows all system components to share a single physical copy of data. It can be used to avoid data copying in network communication. Using IO-Lite, however, would have been difficult as it would have required us to modify network drivers as well as significant parts of the kernel to take advantage of a unified cache and buffer pages. In addition, network streams from different clients must be "early demultiplexed" in order to determine the ACLs of received data objects, incurring extra network protocol processing overhead, which is not required by VRPC.

Bershad et. al. [3] proposed Lightweight Remote Procedure Call (LRPC) to improve the RPC performance on the same machine. LRPC uses a shared stack to reduce copying small or medium size parameters and results between protection domains, and reduces scheduling overhead with the thread tunnelling technique. Applying this scheme to a VM-based NFS poses special challenges. NFS clients frequently issue RPCs to read or write files. The file data is passed by reference in the RPC and can be of large size. With LRPC, if an argument is passed by reference, the referent is copied to the shared stack. Moreover, the shared stack used in LRPC has a fixed size after allocation. This is not a problem for many RPCs, which have small arguments, but it is a problem for RPCs with large arguments. According to the authors, "if data is too large to fit into the argument stack, it must be transferred in a large out-of-band memory segment, which is complicated and relatively expensive". LRPC also relies on a thread tunnelling technique to reduce context switching overhead, but extensive modifications to the virtual machine monitor would be required to make thread tunnelling work across the VM boundary.

Schmidt et. al. [30] proposed to improve the performance of "read-only" RPCs by sharing RPC objects between protection domains within the Opal operating system. Using this technique in the context of distributed file systems has open issues. It would require determining the set of objects that are appropriate to share via read-only RPCs in the context of file system. Read-only RPCs also require the server to not delete or recycle shared objects until a client has stopped using them; otherwise a client could read erroneous data. In the context of file systems, this is a severe constraint. For example, if the server decided to export a read-only inode entry directly from its cache to a client, it would not be able to replace that entry with another inode until it is sure that all clients have released that inode.

2.2 VM Memory Sharing

Virtual machine systems such as Disco [7, 13] and Xen [2] expedite device I/O by sharing memory between VMs. For example, Xen's virtual network driver uses shared memory to avoid data copying. Our proposed mechanisms extend the applicability of inter-VM sharing to higher-level file system operations.

IBM proposes to use a discontiguous saved segments (DCSS) mechanism [15] to allow multiple guest OSes to share common data such as the text segments of common utilities. As such, if a guest OS loads an application that has been previously loaded into shared memory, this guest OS can share the identical text segment, which not only reduces disk I/O, but also saves memory. Similarly, the VMWare ESX server [32] allows pages with duplicate contents to be shared amongst multiple VMs. Identical pages can be located by computing the hash value of page contents or by comparing page contents byte-by-byte. These schemes are complementary to our mechanisms. They can be extended to share identical file data pages across multiple guest OSes, which can improve the scalability of VM-based distributed file systems. On the other hand, our mechanisms, e.g. inter-VM inode sharing, are able to exploit additional data sharing opportunities with file system knowledge, which is not possible with page data comparison.

An ongoing research project called XenFS [33, 34] aims to provide an inter-VM shared file cache for Xen. This work is complementary to our mechanisms. It can unify file caches of different virtual machines and reduce data copying when two VMs read the same file. At present, the work is still ongoing. There are no performance results or design details for further comparison.

2.3 Virtualization Aware File system

Ventana[27], a virtualization aware file system, extends a conventional distributed file system to provide attractive features such as file versioning, fine-grained data sharing, and flexible access control. Ventana primarily focuses on the principles behind a virtual aware file system, and leaves the performance issue as future work. The authors evaluate Ventana mainly with a case study to demonstrate its functional advantages. While the paper does not extensively analyze the performance of Ventana, it briefly mentions that "Ventana's performance is competitive

with other user-level NFS servers in most cases with simple branching." Our scheme can potentially help Ventana to enhance its performance.

3 Background

This section first presents the assumptions that enable an NFS system to take advantage of proposed mechanisms. Next, it reviews Xen's grant table mechanism, which is used extensively for implementing the proposed mechanisms.

3.1 Assumptions

The proposed mechanisms were designed based on the following assumptions of NFS systems that run in a virtual server environment:

1. The file server and the clients reside on *different* virtual machines running on the *same* physical host. They share the same hardware but are separated by software (VMM), which makes it possible to exchange data via shared memory.
2. NFS files are only modified via the NFS server. This assumption is required to ensure consistency of shared inodes. Without it, substantial changes would be required to the file system and generic inode structure. More details are discussed in section 4.3.1
3. The server runs a Linux/Unix file system that uses inodes to describe files. This paper mainly focuses on inter-VM sharing of inodes. It does not cover the scenario where a file system maintains file attributes in different data structures.
4. The server stores files on hard drives instead of other temporary media like RAM or a USB key. Thus, the backing device IDs are relatively stable. It is therefore reasonable to assume that a file can be uniquely identified by its inode number, inode generation number, and backing device number. This assumption is important for inode validation discussed in section 4.3.2.

3.2 Grant Table Mechanism

Grant tables [12] are a mechanism provided by Xen for sharing and transferring memory between virtual machines at the granularity of pages. It allows VM_A to grant VM_B certain rights (read-only or read/write) to specific memory frame(s). Details of the memory sharing procedure are as follows:

1. VM_A grants VM_B access to a specific memory frame. This grant is identified by a grant reference number ref.
2. VM_A transmits the grant reference ref to VM_B.
3. VM_B uses the reference to map the granted foreign memory pages into its address space.
4. After the shared pages are mapped, VM_B can directly access the foreign memory frames.
5. After VM_B finishes accessing the mapped frame, it must unmap the granted frame.
6. After VM_B unmaps the frame, VM_A can safely revoke this grant.

Note that VM_B must unmap the frame before VM_A revokes the grant; otherwise, the corresponding grant access reference cannot be recycled. Because the number of grant access references is limited, excessive failures on recycling grant references will eventually exhaust the reference space and prevent further page sharing. We ran into this problem during our first implementation of the inter-VM metadata sharing mechanism. To cleanly terminate inter-VM memory sharing, we developed a callback mechanism, which is described in Section 4.4.

4 Inter-VM Metadata Sharing

For an NFS client, retrieving file attributes from an NFS server is time-consuming, because it requires exchanging synchronous messages between the NFS client and the server. This cost is even higher for actions such as like listing a directory or compiling a program because each action invokes multiple file attribute fetches.

To reduce communication overhead when retrieving file attributes, NFS caches the retrieved file attributes. A cached file attribute is assumed to be coherent with the server for a short period of time. During this period of time, requests for this file's attributes are served with the cached value. If a cached attribute times out, the NFS client has to re-fetch the file attribute from server before serving subsequent file attribute requests. This incurs significant communication costs to fetch attributes and keep the attribute cache fresh. Moreover, it is often tricky to balance the timeout period and attribute coherence. If timeout period is too long, cached attributes can often be inconsistent with server in a busy scenario, which may cause operation failure. If the timeout period is too short, communication overhead increases and system performance suffers.

We introduce an *inter-VM metadata sharing* mechanism that not only reduces communication times needed for attribute retrieval, but also provides stronger file attribute coherence. The inter-VM metadata sharing mechanism allows an NFS server to open a "window" for the client VM. This "window" exposes specific memory pages that contain NFS file

metadata. An NFS client can then map the exposed pages into its address space and share the mapped metadata with the NFS server. With this mechanism, any metadata updates made by the NFS server are seen by the NFS client immediately. An NFS client can bypass inter-VM communication and read file attributes with local memory accesses, which reduces communication overhead.

While this technique can be generally applied to share various VFS metadata objects such as dentries and superblocks, this paper mainly focuses on the sharing of in-memory inodes [6], because they contain most file attributes needed by an NFS client.

The rest of this section is laid out as follows. Section 4.1 discusses the success conditions of the inter-VM metadata sharing mechanism. Section 4.2 describes how an NFS server grants a client access to an inode stored in its memory space. Section 4.3 illustrates how an NFS client retrieves file attributes from the "exported" inode, including issues of inode consistency and validation. Section 4.4 shows how an NFS server revokes inode sharing grants. Finally, Section 4.5 discusses the security and privacy implications of introducing the inter-VM inode sharing mechanism.

4.1 Conditions for Success

An NFS system running in virtual server environment generally exhibits four ideal properties, which makes it feasible to achieve inter-VM inode sharing:

1. It is possible for an NFS client to directly access a specific memory page of the virtual machine where the NFS server resides. In a virtual server environment, the NFS server and client reside on the same physical host. Their memory frames are on the same hardware. With the help of the virtual machine monitor, it is possible to allow one VM to directly access another VM's memory.
2. Inodes are clustered together. To share an inode, an NFS client must map the memory frame containing this inode into its address space. Mapping memory from another VM incurs certain overhead. If inodes are scattered sparsely in server's memory, the client may have to map a lot of memory pages for inode sharing. The mapping overhead may negate the savings gained from the inter-VM inode sharing. Fortunately, most mainstream file systems allocate inodes from dedicated slab caches [5]. For example, Ext3 allocates inodes from the `ext3_inode_cache` cache; Reiserfs [20] allocates inodes from the `reiserfs_inode_cache` cache. Inodes are therefore contained in limited cache pages and can be mapped at a low cost.
3. File attributes are directly stored in the mapped inodes. Inter-VM metadata sharing is most beneficial to NFS performance if a needed file attribute is in a single page. If file attributes are stored as pointers in inodes, NFS client must map and access multiple pages, which would incur additional costs and render this mechanism less effective. Fortunately, most file attributes needed by NFS are directly stored in inodes. The only exception is the device information of NFS exported directories, which is stored in superblocks and is referred in an inode via a pointer. However, the number of superblocks is limited. The pages containing superblocks can be mapped at an NFS client's initiation stage.
4. Mapped inodes are likely to be valid when they are accessed by NFS clients. By "valid", we mean that a mapped inode is ***not*** freed or filled with another file's inode when it is accessed. The Linux kernel caches inodes when memory is sufficient. Thus, a mapped inode is likely to be valid when system still has available memory. Nevertheless, it is possible that a mapped inode page is shrunk due to memory pressure or a mapped inode is released because of file deletion. To avoid accessing a wrong inode, a mapped inode must always be validated before use. We describe the details of inode validation in Section 4.3.2.

4.2 Exporting Inodes for Sharing

To export an inode for sharing with a client, an NFS server first needs to grant this client access to the page containing the inode. Next, the server must send the client the grant reference as described earlier in section 3.2, which is needed by the client to map the exported inode page. To do this, two questions must be answered:

1. When should the NFS server grant foreign access to an inode page?
2. How should the NFS server send the grant reference to the client?

NFS identifies each file with a persistent file handle. Before an NFS client performs any operations on file, it must first get this file's handle from the server. Therefore, we decide to grant foreign access to a file's inode when the server creates the file handle. The grant information will be carried in the file handle and sent to the client.

To hold inode grant information, in VNFS, we add five fields to NFS file handles:

- `fb_inode_addr` – the the inode's memory address at the server side.

- `fb_inode_ref` – the grant reference of the memory frame where this inode resides.
- `fb_inode_number` – inode number of the file represented by the file handle.
- `fb_inode_generation` – generation number of the inode, which is used to differentiate two files reusing the same inode.
- `fb_inode_devid` – backing device identity.

The last three fields are used for inode validation, which is described in section 4.3.2.

We change the NFS server's `fh_compose()` function to include above fields when NFS file handles are composed. Before composing file handle for a file, the modified `fh_compose()` function first gets the file's inode address `fb_inode_addr`. The inode page address can be easily computed as "$fb\_inode\_addr \& PAGE\_MASK$".

To avoid duplicate export of inode pages, the NFS server maintains an export table for each client. This export table is implemented as a radix tree [23] and is indexed by exported page addresses. By looking up the export table, the hook function can determine whether current inode page has been exported. If the inode page is not exported, the hook function grants the client *read-only* access to the inode page, updates the export table, and sets the `fb_inode_ref` field to the grant reference number. If the inode page is already exported, the hook function simply sets `fb_inode_ref` to the recorded grant reference.

After preparing the above fields, the original `fh_compose()` function is called to compose standard fields of NFS file handle. The composed file handle is sent to the VNFS client as part of results of NFS requests.

4.3 Accessing Shared Inodes

When an NFS client gets an extended file handle, it first extracts the inode sharing information: `fb_inode_addr` and `fb_inode_ref`. With the field `fb_inode_ref`, the NFS client can map the inode page into its address space. After the inode page is mapped, the client locates and validates the inode identified by this file handle. Finally, the client can safely use the shared inode content.

To avoid duplicate mapping, each NFS client maintains a mapping table for each NFS server. The mapping table is a map of the server's entire grant reference space. In the current implementation, it has 4096 entries, which is the maximum number of grant references supported in a VM. In most scenarios, the grant reference space is big enough for inode sharing. When grant reference space is insufficient, the server can request a client to unmap the least referred inode pages to free grant references.

Each mapping table entry indicates whether a specific grant reference has been used to map a page from the server. A mapping table entry mainly contains four fields:

- `ref` – the entry's grant reference
- `ismapped` – whether grant reference `ref` has been used to map an inode page
- `remote_pageaddr` – the inode page address in the server's address space
- `local_pageaddr` – the client side memory address where the inode page was mapped

The mapping table is indexed with the `ref` field.

By looking up the mapping table, the NFS client can determine whether the inode page has been mapped. If the inode page is already mapped, the client skips further mapping steps. Otherwise, the NFS client maps this inode page and updates its local mapping table. After the inode page is mapped, the NFS client uses the file handle combined with the mapping entry to locate the inode identified by the file handle. The mapped inode's client-side memory address is:

$$inode\_addr = (fb\_inode\_addr - remote\_pageaddr) + local\_pageaddr$$

Here, "$fb\_inode\_addr - remote\_pageaddr$" is the offset within the page for the shared inode. With the inode address, the NFS client is able to directly read file attributes from the mapped inode.

4.3.1 Synchronization of Concurrent Access to Shared Inodes

Concurrent access to a shared inode must be carefully controlled to ensure that the fetched inode content is consistent. Suppose the NFS server changes a shared inode while a client is reading file attributes from the same inode. Without proper synchronization, it is possible that some values this client retrieved are stale while the rest are current.

The traditional solution for preventing such a race condition is to use interlocked mutual exclusion [10]. However, in a virtual server, NFS clients and the server reside on different virtual machines. An NFS client cannot modify any data in the server's memory. Therefore, an NFS client cannot acquire a lock from the NFS server without making a RPC to the server. Synchronizing via RPC communication largely negates the benefit of inter-VM inode sharing. Moreover, the NFS server cannot allow an untrusted NFS client to hold a lock because the server can be blocked if the client fails to release the lock properly.

To address this problem, we used the version-based synchronization protocol developed by Schimidt et.

Server	Client
W1. Mutex.lock();	**R1.** *V1=Version[1];*
W2. *Version[0]++;*	**R2.** *take inode snapshot;*
W3. Mutex.unlock();	**R3.** *V0=Version[0];*
W4.	**R4. if (V0 != V1)**
W5. *Update inode;*	**R5.** *fall back to an RPC;*
W6. Mutex.lock();	
W7. *Version[1]++;*	
W8. Mutex.unlock();	

Figure 1: *Concurrent Accesses to Shared Inodes*

al. [30] to guarantee that a fetched inode snapshot is consistent. In this protocol, each shared inode page needs a server-side mutex lock (*Mutex*) and two version numbers (`Version[0]` and `Version[1]`) to mediate concurrent access. Both `Version[0]` and `Version[1]` are one word long, so that reads/writes are atomic to these fields. The NFS server maintains a pair of version numbers for each grant reference in its reference space. All version numbers are located on statically allocated pages globally shared amongst the NFS clients and the server. The version numbers are modified only by the server and synchronized by the mutex lock. For an NFS client, it can use the grant reference associated with a shared inode page to locate the version numbers. These version numbers can only be read by the NFS client to detect concurrent accesses to the inode page.

The concurrent access synchronization mechanism is illustrated in Figure 1. `Version[0]` represents the number of times an update has begun on the inode page, and `Version[1]` counts the number of times an update has completed on the inode page. If the two numbers are equal, then the client can believe that the inode page is in a consistent state; otherwise a server thread concurrently updated some inode on the shared page.

One change we made to the original version-based synchronization protocol is that the mutex lock is only used to protect the page-specific version numbers, and not the inodes to be accessed. This does not break the correctness of the original protocol, because concurrent accesses to a single inode are mediated by the underlying file system. By excluding the inode update code from the locked critical section, this algorithm allows concurrent updates to inodes on a common page.

A VNFS server maintains two version numbers and a mutex lock for each exported inode page. It maintained the version number pairs in a table, called the *Version Number Table*, that needs to be readable by the clients. A client needs to be able to locate the version pair, given a grant reference. But an inode page

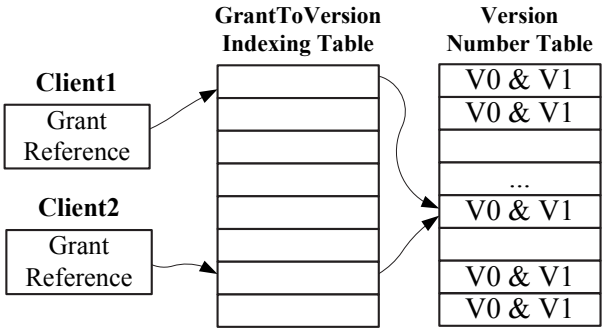

Figure 2: *GrantToVersion Indexing Table. In this figure, two NFS clients share the same inode page. They must use the same version values for getting a consistent inode snapshot*

can be shared by multiple clients. Xen gives each client a different grant reference for this shared page. Therefore, a grant reference cannot be directly used as an index for the table. To address this problem, as illustrated in Figure 2, we used a *GrantToVersion indexing table* to map a grant reference to the right version numbers. Both tables, the GrantToVersion indexing table and the Version Number Table, are located on statically allocated pages globally shared amongst the NFS clients and the server, with clients having read-only access. The maximum number of entries in both tables is set to the maximum number of pages that can be exported from a Xen virtual machine (4096, in our implementation).

To reduce modifications to the OS kernel as much as possible, we assume that all NFS files are accessed via the NFS server. With this assumption, the synchronization mechanism does not require any changes to the linux kernel or the inode structure itself, though it does require changes to the NFS modules. If one wishes to allow NFS files to be modified directly through the server's local file system (e.g., ext3), the guest OS kernel must be changed to update the version numbers properly.

4.3.2 Validating Mapped Inodes

Even if a fetched inode snapshot is consistent, the snapshot itself can still contain false information. For example, a mapped inode can be released due to memory pressure or file deletion. After the mapped inode is released, its memory can either contain stale inode content or be filled with a different file's inode. However, an NFS client has no way to detect changes that occur on the NFS server. Therefore, NFS client must validate a fetched inode snapshot every time before using its content.

The validity of a fetched inode snapshot can be determined by checking two conditions. The first condi-

tion is that the mapped inode is not released. If a file is deleted after its inode is mapped by an NFS client, this file's inode will be released and does not contain valid information anymore. An NFS client can determine whether a mapped inode has been released by checking that inode's `i_state` field. Linux kernel always sets an inode's `i_state` field to "`I_CLEAR`" when releasing an inode. The `i_state` field of a valid inode cannot be equal to "`I_CLEAR`", when accessed from a consistent snapshot.

Second, the mapped inode should be the inode of the file identified by the specified file handle. According to assumption 4 described in section 3.1, a file can be uniquely identified by its inode number, inode generation number, and backing device number. By checking these three fields in the fetched inode snapshot against those stored in the file handle, VNFS clients are able to determine whether a mapped inode is the one identified by the given file handle.

4.4 Inode Page Revocation

When all inodes in an inode page are freed, this page will be released and moved to the free page list of the inode cache for fast reuse. If the free page list already holds enough pages, subsequently released pages must be freed. However, some of these pages may have been mapped by an NFS client, who is unaware that an shared inode page needs to be released by the NFS server's VM. To cleanly free an exported inode page, the NFS server must ask all NFS clients to unmap that page. After all NFS clients unmap the page, the NFS server can safely revoke the grant and recycle the grant reference. This procedure is referred to as "inode page revocation".

We modified the Linux slab manager at the NFS server side to intercept all requests that will free inode pages. If an intercepted page has been exported for sharing, it is put into a "deferred freeing" queue. An NFS server runs a kernel thread to periodically process "deferred freeing" requests in the queue. For each NFS client, the kernel thread scans the "deferred freeing" queue to single out the pages that were exported to that client. For each of such pages, the kernel thread creates an unmapping entry. All unmapping entries are put together to form an "unmapping" request to call back the NFS client. Upon receiving the "unmapping" call, the NFS client first checks all "unmapping" entries against its mapping table to exclude the pages that have not been mapped. Next, the client unmaps the pages that have been mapped. After the "unmapping" call returns, the server can safely revoke the grants and free the pending pages.

A security threat is that malicious NFS client may refuse to unmap inode pages, with the hope of exhausting all grant references and blocking other NFS clients. The effectiveness of this attack can be reduced by specifying a client's quota on grant references. With the grant reference quota, a malicious client refusing to unmap grant references can only exhaust its own reference space. We currently do not use this quota in our implementation, but it may be necessary in some systems, depending on the threat model.

4.5 Security Discussion

A shared inode page could contain inodes of files that are not in directories exported by NFS. Thus, sharing inodes across VMs may cause information leakage. We acknowledge this issue, but believe the threat is mild. First of all, the server exports all inode pages to an NFS client's address space as read-only. This is enforced by the virtual machine monitor. Unless the virtual machine monitor is compromised, NFS clients cannot modify the mapped inode objects. Second, the information leaked from a mapped inode object mainly includes: the inode number, access/modification time, and inode object size. This information is unlikely to be directly used to mount attacks, because it is hard to map an inode to a file without sufficient access rights to the server's file system. If, on the other hand, a user has sufficient access to the server's file system, the file attributes could have been easily obtained via normal applications like "ls" or "stat" without using shared inode pages.

5 VRPC

In a virtual server, NFS clients and server share the same hardware, but they reside in different virtual machines, which are securely partitioned by the virtual machine monitor. To exchange file requests and data, NFS clients and server must rely on cross-machine communication.

Traditionally, NFS uses remote procedure call [4, 9] as a convenient way to exchange data between NFS client and server. However, standard RPC incurs substantial overhead for copying data, traversing network stack, XDR-encoding/decoding data, and establishing connections.

Several NFS implementations have been optimized to avoid data copying by sending file data directly from the kernel page cache to network device layer. However, receivers still need to copy data from network layer to file system buffer. Moreover, the over-

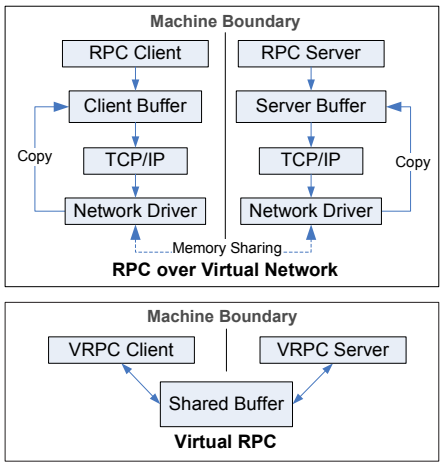

Figure 3: *Comparison of VRPC and standard RPC*

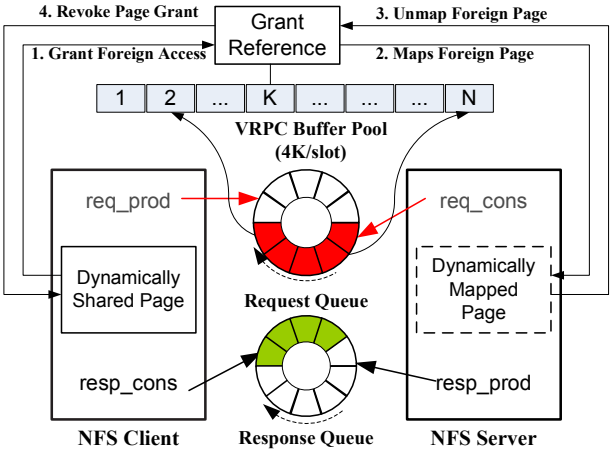

Figure 4: *Architecture of VRPC*

head of protocol processing is nontrivial. For example, if a client receives a read response, it must put the received data blocks into the file cache. To do this, the network interface first strips off transport headers and the NFS header from each message. Then, the network interface must assemble the packets and put them into page-aligned file buffer. If the network MTU is smaller than a hardware page, a page of data can be spread across multiple packets, which can arrive at the receive out of order or mixed with packets from other flows.

To speedup RPC communication, this paper presents the *Virtual Remote Procedure Call (VRPC)* mechanism which can significantly reduce the number of times that data has to be copied and eliminate the overhead of a network protocol stack. A VRPC has the same format as a normal remote procedure call. However, it uses cross-VM memory sharing to achieve fast data exchange between virtual machines running on the same hardware, completely bypassing the network stack.

At the initialization stage, VRPC establishes a shared memory region between the NFS server and each NFS client. Any requests or responses can be directly put into the shared memory and be seen by the receiver immediately. There is no need to copy, encode, fragment or encapsulate data, which greatly reduces the data processing overhead. In addition, VRPC eliminates the network connecting/disconnecting cost since it does not establish a network connection at all.

Figure 3 compares standard RPC and VRPC. The top half shows RPC running over Xen virtual network. Xen already uses inter-VM memory sharing to expedite network transmission. However, RPC running over Xen virtual network still needs to pay the overhead of protocol processing and data copying from network stack to VRPC buffer or file cache. In contrast, VRPC, as shown in the bottom half of Figure 3, allows client and server to directly access data via shared memory, eliminating the overhead of protocol processing, packet assembly, and data copying.

An example helps illustrate the overall working of VRPC. Suppose an NFS client wants to read a page of file data from the NFS server. It first allocates a page from the file cache to store file data. Then, the client grants the server read/write access to this page. Next, the client creates a VRPC request which contains the file handle, offset, bytes to read, and the page sharing information. Finally, the client deposits the VRPC request into the shared memory. Upon detecting this request, the server puts file data directly to the client's file cache page. No additional memory copy or network transmission occurs in this procedure.

In the rest of this section, we first present the hybrid memory sharing model employed by VRPC. Next, we describe how VRPC requests and responses are organized. Finally, we illustrate the mechanism that VRPC uses to notify VMs of outstanding requests or responses.

5.1 Hybrid Memory Sharing

Different NFS requests may need VRPC to carry different amount of data. For example, function *lookup()* searches certain directory for a specified file. The input parameters are the file name and the parent directory's file handle. The output values are the file's handle and attributes such as file size, modification time, and access rights. This function only involves small amount of data exchange. On the other hand, functions such as *read()* or *write()* can exchange data of arbitrary length.

VRPC uses a hybrid memory sharing mechanism to carry variable length of data with minimum over-

head. Small amount of data is directly passed via a statically shared VRPC buffer pool. Bulk data is exchanged via dynamically shared memory regions.

As shown in Figure 4, both an VRPC client and the server share a memory region called the *VRPC Buffer Pool*. An VRPC client allocates a VRPC buffer pool at its initialization stage. The VRPC server then maps the buffer pool into its address space. For the rest of its life, the NFS client can use this buffer pool to exchange data with the server. VRPC buffer pool only requires an NFS client to pay one-time memory mapping cost during its whole lifetime, which is very cheap.

A VRPC buffer pool is statically divided into multiple buffer slots, with each buffer slot corresponding to a VRPC request or response. The size of buffer slot is fixed but can be configured. Our current implementation sets each slot to be 4K byte long. VRPC buffer pool is designed to carry small amount of data at minimum cost. ALL NFS requests, except ***read(), write(), readlink() and readdir()*** functions, can pack their request arguments and results directly into a buffer slot(less than 4K bytes).

For functions that need to exchange data of variable length or more than 4K bytes of data transfer, VRPC allows an NFS client to dynamically allocate memory and grant the server access to this memory region. The NFS client deposits request arguments in the dynamically allocated memory, but put the sharing information, including grant references, the starting address and size of the memory, into the buffer pool slot. With the sharing information transmitted via the shared buffer pool slot, the VRPC server maps the dynamically shared memory into its address space for data accessing. Currently, VNFS only uses dynamic memory sharing during four NFS functions: ***read(), write(), readlink() and readdir()***. All these functions involve variable-length data exchange. Dynamic memory sharing is slightly more expensive, as it needs to establish memory mapping for each file request that needs dynamic mapping. But dynamic memory sharing allows arbitrary length of data to be transferred with a VRPC call.

5.2 VRPC Request/Response

As illustrated in Figure 4, VRPC uses two circular queues to coordinate requests and responses between two VMs. Both of these two queues are divided into multiple fixed size slots (15 byte per slot in current implementation), with each request or response using one slot. NFS client puts file requests into the request queue and fetches responses from the response queue. NFS server fetches the requests from the request queue and puts the responses back to the response queue. A VRPC request and its response carry an identical, unique VRPC ID. A VRPC client uses the VRPC ID to match a received response with its corresponding request. With this design, multiple RPCs can be in-flight and responses can be delivered out of order. This allows a client to have multiple operations with pending reads and writes to be in progress.

Each VRPC request is associated with a VRPC buffer pool slot. A client uses the buffer pool slot to directly store request arguments if they can fit into this slot. If a VRPC request needs to exchange data larger than a buffer pool slot, the client uses dynamic memory sharing mechanism. Under such a condition, the client uses the buffer pool slot to store the following: the grant reference to the dynamic page(s), the starting address, and the size of the dynamically allocated memory. The server uses these data to share the dynamic memory region. After processing a request, the VRPC server deposits result data into the buffer pool slot associated with this request. Therefore, a client cannot free a request's buffer pool slot until the response is received and consumed.

In summary, a VRPC request is composed of two parts: request metadata and request arguments. The request metadata mainly includes VRPC ID and the index of the associated buffer pool slot. They are put in a request queue slot. The client puts the request arguments either in the associated buffer pool slot, or in a dynamic memory region which is described by the buffer pool slot associated with the request.

5.3 Request/Response Notification

When a new request or response is deposited, its producer must notify the other party that new data is outstanding. VRPC uses Xen's event channel mechanism to do this notification. Event channels are an asynchronous inter-VM notification mechanism. When a new NFS client is started, the NFS server instantiates an event channel between the NFS client's VM and itself. Both virtual machines can request that a virtual IRQ be attached to notifications on a given channel. The result of this is that one virtual machine can send a notification on an event channel, resulting in an interrupt in the other virtual machine. The event channel can only transmit a one bit message, so it is not suitable for transferring bulk data, but it is reasonable for event notification. After attaching a virtual IRQ to the event channel, both virtual machines install IRQ handlers to handle messages sent over the event channel. An event detected on the event channel indicates that a new request or

Hardware	
CPU	3.00GHz Pentium IV
Memory	512MB(Dom0) 512MB(DomU)
Disk	Maxtor 7200RPM EIDE
Software	
VMM	Xen 3.0.2
Domain0 OS	Linux 2.6.16-xen0
DomainU OS	Linux 2.6.16-xenU
Linux Distribution	Fedora Core 4
Postmark	version 1.5
Tar	version 1.15.1
GNU gcc	version 4.0.2
GNU ld	version 2.15.94.0.2.2
GNU Autoconf	version 2.59
GNU automake	version 1.9.5

Table 1: *Experimental platform*

response is put into the shared communication ring. Upon receiving a new notification message, the server or client checks the shared queues to get new requests or responses.

6 Evaluation

We ran all experiments on a machine whose configuration is listed in Table 1. We deployed the NFS server in the privileged guest VM (Domain0, in Xen terminology), and the NFS client in an unprivileged guest VM (DomainU). Both Domain0 and DomainU used Ext3 as the local file system. DomainU used two virtual disks as the Ext3 backing device. The experimental NFS directory was exported and mounted with "`ASYNC`" option in all test runs.

All benchmarks were run on four file system configurations:

- A virtual disk that uses the Ext3 file system
- A standard NFS client and server that use Xen's virtual network
- Two versions of VNFS, a limited version that only uses VRPCs and a full version that uses both VRPCs and inter-VM metadata sharing.

We conducted all measurements in DomainU.

For NFS based tests, we copied all directories and files on the experimental machine to an NFS exported directory. Before running a benchmark, we always "`chroot`" [21] to the NFS mounted directory. As a result, all input and output files needed for the benchmark are accessed from the tested NFS file system. We also directed console outputs to "/dev/null" to exclude latency caused by console printing.

In all experiments, the network between virtual machines is an internal-bridged virtual network provided by Xen. Because Xen's virtual network transfers data at best effort, we do not control or limit its network bandwidth. To avoid warm cache effects caused by previous runs, we unmounted both NFS client and server-side file systems after each round of benchmark. We executed all experiments right after the file system was mounted. Each experiment was run 10 times and the reported results reflect 90% confidence interval.

6.1 Apache Build

We first used Apache build as a representative of typical workloads on a normal development machine. We used Apache 2.0.58 as the benchmark object. The Apache archive includes 2339 files scattered in 188 directories. The total size of the archive is 6.13MB before being decompressed. After being decompressed, the total size of the Apache directory is 32.9MB. The benchmark first `unpacks` the archive of Apache 2.0.58 into a source directory. Next, it runs `configure` to build the source code dependency, which involves lots of small data read and file lookups. During the third phase, it `builds` the Apache binaries from the source files, which is a CPU intensive task, but also generates lots of object files and temporary files. Finally, it `removes` all Apache files including the Apache source tree, generated configuration files, object files, and Apache executable binaries.

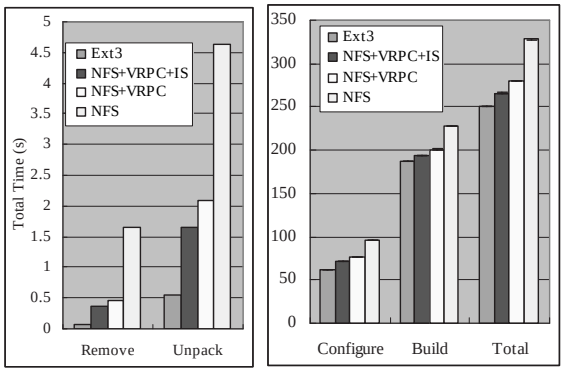

Figure 5: *Performance of Apache build workload. "NFS+VRPC+IS" stands for NFS with VRPC and Inode Sharing mechanisms enabled;"NFS+VRPC" stands for NFS with only VRPC mechanism enabled; "NFS" stands for standard NFS over virtual network*

In Figure 5, each bar group shows a phase of the Apache build benchmark, while the "Total" group represents the total time consumed in the four phases of the benchmark. Overall, the standard NFS running over Xen virtual network was 31% slower than Ext3 file system. In contrast, VNFS enhanced with the VRPC and inter-VM inode sharing techniques was only 6.3% slower.

By examining the benchmark results more deeply, we found that the performance speedups were high in `unpack` and `remove` phases. The speedup percentages were 64% and 77%, respectively. The effect of inter-VM inode sharing was also pronounced during these two phases. NFS with VRPC and inter-VM inode sharing outperformed NFS with only VRPC by 21%.

In the `configure` phase, VRPC and inode sharing techniques boosted NFS performance by 26% and saved 25 seconds. In the `build` phase, VRPC and inode sharing techniques improved NFS performance by 15% and saved 35 seconds. In `configure` and `build` phases, NFS with both VRPC and inter-VM inode sharing support ran faster than NFS with only VRPC support by 7% and 4%, respectively. Since computation represents a large portion of the `configure` and `build` phases, reduced impact of file system improvement was expected. Nevertheless, the improvements were still 26% and 15% over standard NFS.

6.1.1 Performance of File System Calls

The total time of Apache build includes the time used for file accessing as well as time spent on system computation (e.g. compiling). To better evaluate how the proposed mechanisms impact file system operations, we collected the latencies of major file system calls during execution of the workload.

Table 2 lists a summary of results. "VNFS" represents the VNFS with VRPC and inter-VM metadata sharing enabled. "NFS" represents standard NFS running over Xen virtual network. The "Speedup" row shows the improvement provided by VNFS over NFS.

With the proposed mechanisms, six out of the eight file system operations improved performance by over 50%. The exceptions were `read()` and `write()` operations. The reason is NFS cache and the file accessing pattern in the Apache build workload. An NFS client always tries to take advantage of its local cache to reduce communication. The `read()` requests to the files that have been loaded to the local cache are served from the local cache. During execution of Apache build workload, a source file can be read multiple times. This file accessing pattern limits overall improvement on file reads.

Similarly, when a process writes to an NFS file, NFS client will cache the dirty pages at the local cache and defer flushing them to the server as late as possible. If memory is sufficient, the NFS client can defer sending dirty pages to the server until the file will be closed. Therefore, the performance improvement observed on `write()` system calls are smaller.

However, for the same reason, close() operations are 65% times faster with the proposed mechanisms. When a file is to be closed, an NFS client must flush all of this file's dirty pages to the server to guarantee "open-to-close" semantics of NFS. Therefore, the `close()` operations can require substantial cross-VM communication.

6.2 Postmark Benchmark

We next ran the Postmark (version 1.5) benchmark, which was designed to simulate small-file workloads seen in electronic mail, netnews, and web-based commerce [16].

In each run, we configured Postmark to create 10000 files and perform 10000 transactions consisting of file reads, writes, creates, and deletes, and removal of all files.

We run four groups of benchmarks. In group 1, we used the following default Postmark setting: the file size range is from 512 bytes to 9.7 kilobytes; the probabilities of read and write operations are equal.

In group 2, we used the default file size range, 512 bytes to 9.7 KB, but set the probability of read operations to be four times of that of write operations. This is closer to the workload of some web sites in which most accesses are read-only.

In group 3 and 4, we increased the upper bound of file size range from 9.7KB to 40KB, to analyze how our system performs on files of larger size range. Postmark was developed in 1997, with increasing network bandwidth and richer content, web sites and email systems often access larger files. Group 3 and 4 used the same read/write ratios as group 1 and 2, respectively.

As Figure 6 shows, VNFS significantly improved on NFS performance in all four groups. When the file range is 512 bytes to 9.7K bytes, VNFS outperformed standard NFS by 46%. When file range is 512 bytes to 40K bytes, VNFS ran 37% faster than standard NFS.

The speedup percentage is higher when tested files are smaller. With larger files, longer disk read/write times mask part of benefits gained with the inter-VM inode sharing and VRPC techniques. We did not observe a significant difference in percentage improvement when reads dominate writes.

The results also show that Ext3 clearly outperformed both VNFS and NFS on the Postmark benchmark. There are several reasons for the performance difference. One reason is that VNFS and NFS need to cross VM boundary to access files, which incurs additional context switch overhead. Another reason is that Ext3 and NFS have different semantics. Ext3

		stat64	fstat64	utime	open	read	write	close	unlink
Time/Call (ms)	Ext3	3.47	0.82	9.50	3.84	89.66	10.93	1.40	16.75
	VNFS	9.32	1.89	56.22	10.78	99.40	7.16	5.15	78.25
	NFS	20.15	3.89	145.81	69.61	162.69	8.16	14.51	172.90
Speedup of VNFS over NFS		54%	52%	61%	85%	39%	13%	65%	55%

Table 2: *Latencies of major file system calls collected during Apache build workload*

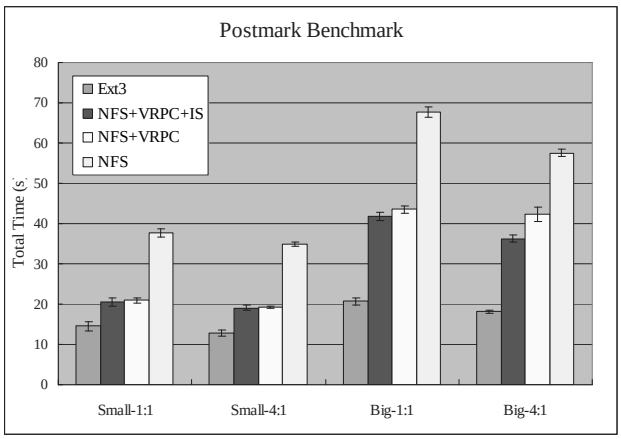

Figure 6: *Postmark benchmark results. "Small" means small file size range (512B-9.7KB), while "big" means larger file size range (512B-40KB). Ratio "1:1" means that the probabilities of read and write operations is equal. Ratio "4:1" means that the probability of read operations is four times of that of write operations*

allows lazy data flush: if a process writes some data to a file and then closes the file, Ext3 buffers the written data in page cache and immediately returns from `close()`. The dirty pages will be flushed to disk asynchronously. For both VNFS and standard NFS, when a file is closed, the client must copy all cached dirty pages to the server to enforce *close-to-open* semantics before returning. While data exchange over VRPC is more efficient than over a virtual network, the data flushing still substantially impacts file system performance.

6.3 Discussion

Our current implementation adds 4300 lines of code to the NFS client module and 3800 lines of code to the NFS server module.

For implementing inter-VM inode sharing, in order to cleanly revoke inode page grants, we added 60 lines of code to the guest Linux kernel at the NFS server side to intercept all requests that free inode pages. In addition, to avoid adding version-based synchronization mechanism to the guest kernel, we decided to impose a restriction that NFS files should only be modified via the NFS server. In practice, this is likely to be a modest imposition. If the NFS server is not running or not exporting any files, it would be safe to access the files directly via the local file system on the server. If it is running, an administrator guest machine can be used to modify the files. Given the significantly lower overheads with our scheme, we believe that should be acceptable.

VRPC is independent of inter-VM inode sharing mechanism and does not require any modification to the guest OS kernel; only changes are to the NFS modules. If kernel change is not acceptable, NFS can still use VRPCs to enhance performance. Furthermore, if a system only uses VRPCs, it would be safe to allow writes to the files on the server via the local file system. Our results indicate that this would still provide substantial benefits from a performance perspective.

Other performance improvements are possible that we hope to consider in the future. For example, because the client and the server maintain separate data caches, data must still be copied between these caches. Using ideas from IO-lite [24] and XenFS [33, 34], it may be possible to reduce or eliminate such data copies.

The Xen system currently imposes a limit on the number of grant references for each virtual machine. If the limit is low and the number of clients is high, inter-VM inode sharing will be less beneficial to performance because the number of pages that a client can share will be limited by the grant reference space. We did not explore this issue in our study. From our interactions with Xen developers, it appears the Xen community is working on finding ways to remove or to significantly increase this limit.

7 Conclusion

This paper explored mechanisms that can improve performance of distributed file systems when the file server and clients reside in different virtual machines on the same physical host. We presented an in-depth design of two mechanisms: inter-VM metadata sharing and VRPC to help reduce overheads of such systems. Both these two mechanisms use shared memory to reduce communication overhead without compromising VM-imposed boundaries.

We implemented these two mechanisms on the Xen virtual machine platform and adapted NFS version 3 to make use of them. The modified NFS resulted in

a substantial improvement over standard NFS.

We discussed solutions to handling consistency of file system data that is accessed via shared memory. VRPC requires no changes to the virtual machine monitor or guest kernels (only changes are to the NFS modules). Inter-VM metadata sharing requires localized changes to the server's guest kernel.

As performance of distributed file systems within virtual machines is improved, extended distributed file systems can work as a practical storage medium for guest virtual machines. Our future plans include exploring additional opportunities for performance improvement and to extend virtual machine file systems to provide features such as fine-grain access control and automatic versioning to multiple guests.

References

[1] Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, November 2002. http://www.lustre.org/docs/whitepaper.pdf.

[2] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[3] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 102–113, New York, NY, USA, 1989. ACM Press.

[4] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles*, page 3, Bretton Woods, NH, 1983. Association for Computing Machinery.

[5] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceeding of USENIX Summer 1994 Technical Conference*, pages 87–98, 1994.

[6] D. P. Bovet and M. Cassetti. *Understanding the Linux Kernel (Ed. A. Oram)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[7] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[8] B. Callaghan, T. Lingutla-Raj, A. Chiu, P. Staubach, and O. Asad. Nfs over rdma. In *NICELI '03: Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence*, pages 196–208, New York, NY, USA, 2003. ACM Press.

[9] B. Callaghan, B. Pawlowski, and P. Staubach. RFC1813: NFS version 3 protocol specification, June 1995. Informational RFC.

[10] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Commun. ACM*, 14(10):667–668, 1971.

[11] M. DeBergalis, P. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle. The direct access file system. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 175–188, Berkeley, CA, USA, 2003. USENIX Association.

[12] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Reconstructing I/O. Technical Report UCAM-CL-TR-596, University of Cambridge, Computer Laboratory, Aug. 2004.

[13] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):229–262, 2000.

[14] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[15] IBM Corporation. How to use Execute-in-Place Technology with Linux on z/VM. Technical Report SC33-8287-00, IBM Corporation, Oct,2005.

[16] J. P. Katcher. Postmark: A new file system benchmark. Tech. Rep. TR3022, Network Appliance, 1997.

[17] K. Kleinpaste, P. Steenkiste, and B. Zill. Software support for outboard buffering and checksumming. In *SIGCOMM*, pages 87–98, 1995.

[18] K. Magoutis. Design and implementation of a direct access file system (DAFS) kernel server for FreeBSD. In *Proceedings of the BSDCon 2002*, pages 65–76, Berkeley, CA, USA, 2002. USENIX Association.

[19] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and performance of the direct access file system. In *Proceedings of USENIX 2002 Annual Technical Conference, Monterey, CA*, pages 1–14, June 2002.

[20] C. Mason. Journaling with ReisersFS. *Linux J.*, 2001(82es):3, 2001.

[21] R. McGrath and Free Software Foundation. Chroot c run command or interactive shell with special root directory. The Linux Manual Pages.

[22] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: a distributed personal computing environment. *Commun. ACM*, 29(3):184–201, 1986.

[23] D. R. Morrison. PATRICIA-Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[24] V. S. Pai, P. Druschel, and W. Zwaenepoel. Io-lite: a unified i/o buffering and caching system. *ACM Trans. Comput. Syst.*, 18(1):37–66, 2000.

[25] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceeding of USENIX Summer 1994 Technical Conference*, pages 137–152, 1994.

[26] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. *Proceedings of the 2nd international system administration and networking conference (SANE2000)*, page 94, 2000.

[27] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *3rd Symposium of Networked Systems Design and Implementation (NSDI)*, May 2006.

[28] R. Sandberg. The Sun Network Filesystem: Design, Implementation, and Experience. In *Distributed Computing Systems: Concepts and Structures*, pages 300–316. IEEE Computer Society Press, Los Alamos, CA, 1992.

[29] M. Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 20–21, 1990.

[30] R. W. Schmidt, H. M. Levy, and J. S. Chase. Using shared memory for read-mostly rpc services. In *HICSS '96: Proceedings of the 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture*, pages 141–149, Washington, DC, USA, 1996. IEEE Computer Society.

[31] T. Ts'o and S. Tweedie. Planned extensions to the Linux Ext2/Ext3 filesystem. In *Proc. of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 235–244, 2002.

[32] C. A. Waldspurger. Memory resource management in vmware esx server. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 181–194, New York, NY, USA, 2002. ACM Press.

[33] M. Williamson. 1st year progress report. http://www.cambridge.intel-research.net/ mwilli2/proposal_final.pdf.

[34] M. Williamson. Xen wiki: Xenfs. http://wiki.xensource.com/xenwiki/XenFS.

[35] X. Zhao, K. Borders, and A. Prakash. Towards protecting sensitive files in a compromised system. In *3rd International IEEE Security in Storage Workshop*, 2005.