VirtFS—A virtualization aware File System pass-through

Venkateswararao Jujjuri
IBM Linux Technology Center
jvrao@us.ibm.com

Eric Van Hensbergen
IBM Research Austin
bergevan@us.ibm.com

Anthony Liguori
IBM Linux Technology Center
aliguori@us.ibm.com

Badari Pulavarty
IBM Linux Technology Center
badari@us.ibm.com

Abstract

This paper describes the design and implementation of a paravirtualized file system interface for Linux in the KVM environment. Today's solution of sharing host files on the guest through generic network file systems like NFS and CIFS suffer from major performance and feature deficiencies as these protocols are not designed or optimized for virtualization. To address the needs of the virtualization paradigm, in this paper we are introducing a new paravirtualized file system called VirtFS. This new file system is currently under development and is being built using QEMU, KVM, VirtIO technologies and 9P2000.L protocol.

1 Introduction

Much research has focused on improving virtualized disk and network device performance, and indeed, modern hypervisors like KVM have been able to obtain good performance as a result and are now a viable alternative to bare metal systems. Through the introduction of higher level interfaces for guests to interact with a hypervisor, we believe it is possible to obtain better performance than bare metal in consolidated workloads [8].

This paper explores an example of one such interface, VirtFS. VirtFS introduces a paravirtual file system driver based on the VirtIO [13] framework. This interface presents a number of unique advantages over the traditional virtual block device. The majority of applications (including most hypervisors) prefer to interact with an Operating System's storage API through the file system interfaces instead of dedicated disk storage APIs. By paravirtualizing a file system interface, we avoid a layer of indirection in converting guest application file system operations into block device operations and then again into host file system operations.

In addition to performance improvements over a traditional virtual block device, exposing guest file system activity to the hypervisor provides greater insight to the hypervisor about the workload the guest is running. This allows the hypervisor to make more intelligent decisions with respect to I/O caching and creates new opportunities for hypervisor-based services like de-duplification.

In Section 2 of this paper, we explore more details about the motivating factors for paravirtualizing the file system layer. In Section 3, we introduce the VirtFS design including an overview of the 9P protocol, which VirtFS is based on, along with a set of extensions introduced for greater Linux guest compatibility.

Section 4 describes the implementation of VirtFS within QEMU and within the Linux Kernel's v9fs file system. Section 5 presents our initial performance evaluation.

2 Motivation

Virtualization systems have historically focused on providing the illusion of access to underlying hardware devices (either by emulating the physical device interface or through a paravirtualization API). Within Linux, the de facto standard for paravirtual I/O communication is the VirtiIO subsystem. At the time of this writing, the mainstream Linux kernel had VirtIO devices for console, disk, and network as well as a PCI passthrough device and a memory ballooning device.

2.1 Paravirtual Application and System Services

What has been largely ignored within the mainstream virtualization community is the opportunity for raising the level of interface from the physical device layer to higher-level system and even application services. Such an approach has been used within academic and research communities in the implementation of microkernels [1], exokernels [4], multikernels [14], and satellite kernels [9]. Enabling paravirtualization of application and system services can provide a hybrid environment leveraging the security, isolation, and performance properties of microkernel-class systems within more general purpose operating systems and environments.

There are a number of reasons favoring the use of paravirtualized system services versus emulated or paravirtualized devices. One of the more compelling arguments is a higher degree of information available about what the guest system (or the guest system's user) is trying to do. For instance, within a desktop configuration, the hypervisor can provide a frame buffer device for graphics, but if we move the interface up a level we can provide paravirtualized access to the hosts windowing system (whether that windowing system is implemented by the operating system or an application infrastructure). This would allow applications within the guest to open new windows on the user's desktop versus within a frame buffer on the desktop. The difference is perhaps subtle in this example, but provide a dramatically different experience from the user's perspective.

Similarly, instead of virtualizing the network device, the hypervisor can provide a paravirtualized interface to the host IP stack. This eliminates complex configuration issues surrounding setting up network-address-translation, bridging devices, and so forth. Guest applications using sockets just use the host's TCP/IP stack directly. Such an approach works well for services which the host is already managing multiplexing for multiple applications (such as graphics, networking, audio, etc.).

An alternative to providing a paravirtualized server would be to use an existing distributed resource access protocol (such as X11 for graphics, PPTP for networking, etc) over a virtualized network device. Such an approach encounters a number of problems. First, it requires both the host and the guest have configured networks, and that the servers and clients be configured to connect to each other. Assuming that you aren't using a dedicated virtual network device for this communication you then incur a number of security concerns and potential sources for performance interference. Assuming one solves all of those problems, there is also the issue of the additional overhead of encapsulating service requests in TCP/IP which is completely unnecessary considering it is very unlikely that you will drop packets between guests and host, with much more effective flow-control being capable in such a tightly coupled system.

By contrast, a paravirtual interface provides a dedicated (performance and security isolated) channel, preconnected between guest and host (no configuration necessary), which doesn't incur any of the overheads of arbitrary and unnecessary encapsulation which going over a network (particularly a TCP/IP network) incurs. Additionally, the tighter binding of a paravirtual API may allow sharing optimizations and management simplification which is unavailable on either a device based or network based interface.

2.2 Paravirtual File Systems

File systems are a particularly good target as a paravirtual systems service. In addition to the points made above, the properties of how file systems are used, managed, and optimized by an operating system make them ideal candidates.

One of the principle problems with virtualized storage in the form of virtual disks is that data on the disk can not be concurrently accessed by multiple guests (or indeed even by the host and the guest) unless the disk is read-only. This is because of the large amount of in-memory state maintained by traditional disk file systems along with the aggressive nature of the Linux dcache and page cache. Read/write disk access can only be accomplished through exclusive access or when negotiated by a secondary protocol. The Linux storage caches introduce another level of inefficiency, caching independent copies of the disk block on both the host and the guest.

If we use a traditional distributed file system (such as NFS or CIFS) over a virtualized network device to access storage, we run into the configuration, management, and encapsulation overheads mentioned previously. We also encounter problems with two management domains for the purposes of user ids, group ids, ACLs, and so forth. The distributed file systems also incur the double-cache behavior of the virtual disks. The other problem is that many distributed file systems impose their own file system semantics on operations, which may be different from the behavior expected from a disk file system.

2.3 Application Use Cases

Perhaps the most straightforward application of a paravirtualized file system is to replace the virtual disk as the root file system. When the system boots, the bundled ram disk simply connects to an appropriately labeled VirtFS volume to retrieve the root volume. Such an approach would allow host-based stackable file systems to be used for rapid cloning, management, and update [10].

In addition to root file systems, the paravirtual file system can be used to access a shared file system coherently from several guests. It can also be used to provide guest to guest file system access. Additionally, it could be used to access synthetic file systems on the host or other guests in order to access management and control interfaces.

Another use case, which we utilized as part of the Libra [2] and PROSE [15] projects is to provide file system offload API for applications running within a LirbaryOS. In such instances, the application is only running on top of a very thin OS interface which doesn't itself contain system services such as a disk file system or network stack. These services are obtained remotely through forwarding I/O requests (at a file system or socket level) to a server running on the host.

Finally we would like to refer to the use cases of the cloud environment. In a cloud environment where multiple guests share resources on a host, portions of the host file systems can be exported and VirtFS mounted on guests giving a secure window of host file systems on the guest. This gives guests more like a local file system interface to portions of host file system. VirtFS also could be very useful to provide a secure way to provide storage services to different customers from a single host filesystem. This also helps to take advantage of various file system features like de-dup, snapshots etc.

3 Design

VirtFS provides functionality that is somewhat similar to a traditional network file systems (NFS/CIFS). The QEMU server elects to export a portion of its file system hierarchy, and the client on the guest mounts this using 9P2000.L protocol. Guest users see the mount point just like any of the local file systems, while the reads and writes are actually happening on the host file system.

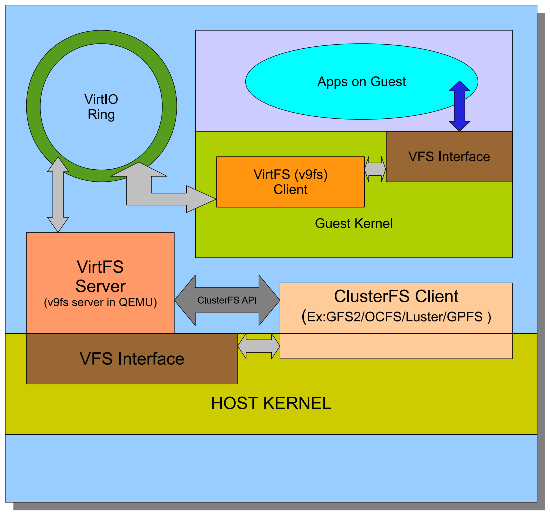

Figure 1: VirtFS block diagram

Figure 1 shows the high level VirtFS anatomy. Server is part of QEMU and client is part of the guests kernel. The protocol is exchanged between the client and the server over VirtIO transport (section 3.5). Server running in the user mode opens up potential for direct communication with the fileserver API (section 6.4).

The major difference between a traditional network file system and VirtFS is its simplicity and optimization to the KVM environment. Our approach is to leverage a light-weight distributed file system protocol directly on top of a paravirtualized transport in order to remote offload elements of the Linux VFS API to a server run on the virtualization host (or within another partition). In order to expedite implementation, we selected a preexisting distributed file system which closely matched our desired approach 9P, and provided a virtualization specific transport interface for it through VirtIO. We are extending the 9P protocol to better match the Linux VFS API. The 9P protocol [3], originally developed as part of the Plan 9 research operating system from Bell Labs [11], and incorporated as a network based distributed file system within the mainline Linux kernel during the 2.6.14 release [6].

3.1 Plan 9 Overview

Plan 9 was a new research operating system and associated applications suite developed by the Computing Science Research Center of AT&T Bell Laboratories (now a part of Lucent Technologies), the same group that developed UNIX , C, and C++. Their intent was to explore potential solutions to some of the shortcomings of UNIX in the face of the widespread use of high-speed networks to connect machines. In UNIX, networking was an afterthought and UNIX clusters became little more than a network of stand-alone systems. Plan 9 was designed from first principles as a seamless distributed system with integrated secure network resource sharing.

Plan 9s transparent distributed computing environment was the result of three core design principles which permeated all levels of the operating system and application infrastructure:

1. develop a single set of simple, well-defined interfaces to services
2. use a simple protocol to securely distribute the interfaces across any network
3. provide a dynamic hierarchical structure to organize these interfaces

In Plan 9, all system resources and interfaces are represented as files. UNIX pioneered the concept of treating devices as files, providing a simple, clear interface to system hardware. Plan 9 took the file system metaphor further, using file operations as the simple, well-defined interface to all system and application services. The benefits and details of this approach are covered in great detail in the existing Plan 9 papers [12].

3.2 9P Overview

9P represents the abstract interface used to access resources under Plan 9. It is somewhat analogous to the VFS layer in Linux. In Plan 9, the same protocol operations are used to access both local and remote resources, making the transition from local resources to cluster resources to cloud resources completely transparent from an implementation standpoint. Authentication is built into the protocol, and the protocol itself may be run over encrypted and digested transports. It is important to understand that all 9P operations can be associated with different active semantics in synthetic file systems. Traversal of a directory hierarchy may allocate resources, or set locks. Reading or writing data to a file interface may initiate actions on the server, such as when a file acts as a control interface. The dynamic nature of these synthetic file system semantics makes caching dangerous and in-order synchronous execution of file system operations a desirable default. For an example of the potential difficulties, think of interacting with the proc or sysfs file systems in Linux over a cached file system mount where some of the data can be dozens of seconds out of date with the actual state on the server.

The 9P protocol itself requires only a reliable, in-order transport mechanism to function. It is commonly used on top of TCP/IP, but has also been used over RUDP, PPP, and over raw reliable mechanisms such as the PCI bus, serial port connections, and shared memory.

The base 9P protocol is based on 12 paired protocol operations (each with a request and response version) and a special error response which is used to notify the client of problems with a request. They are made up of protocol accounting operations (version, authentication, attach, flush), file operations (lookup, create, open, read, write, remove, close), and meta-data operations (set attributes, get attributes). The nature of the operations is strict balanced RPC (each request gets a single response). The protocol is stateful, requiring the server to maintain certain aspects of the state of session and outstanding transactions. The protocol has provisions for supporting multiple outstanding transactions on the same transport and can be run in synchronous or asynchronous modes (depending on the implementation of the client and the server).

As mentioned previously, the 9P file system was added to the 2.6.14 mainline linux kernel with support for TCP/IP and named pipe transports (such that it could be used to access user-space file systems). Later, in 2.6.x the transport interfaces were modularized to allow for alternative transports to be plugged into the 9P client. In 2.6.x an RDMA interface was added by Tom

Tucker support infiniband, and in 2.6.x fscache support was added by Abhishek Kulkarni. It currently has the distinction of being the distributed file system implemented with the fewest lines of code in the Linux kernel.

3.3 9P Extensions

The 9P protocol was originally designed specifically for the Plan 9 operating system, and as such implements a subset of the rich functionality of the Linux VFS layer. Additionally, it has slightly different definitions for certain parameters (such as permissions, open-mode, etc.). It also takes a different approach to user and group management, using strings instead of ids.

In order to adapt 9P to better supporting POSIX during the Linux port, the protocol was extended with the 9P2000.u (for unix) version. This protocol version provided support for numeric user and group ids, provided additional mode and permission bits as well as an extended attributes structure in order to be POSIX compliant, and augmented existing operations to support symlinks, links, and creation of special files.

The .u protocol has been used for the past several years, but had a number of deficiencies. It did not include full support for Linux VFS operations. Notably absent was support for quotas, extended attributes, and locking. Furthermore, its overloading of existing operations to provided extended functions made developing servers and clients to support 9p2000.u problematic, and no partial support for 9p2000.u was possible due to differences in the size of operation protocol messages and ambiguous definitions of extension fields.

In response to these deficiencies, a new approach to offering extensions to the protocol was developed. While core protocol elements (such as size prefixed packets, tagged concurrent transactions, and so forth) remain the same, extended features will get their own operations in a complimentary op-code space to the existing prtoocol. The Linux binding for 9P, titled 9P2000.L, will be the first approach at such an extension and will be aimed at addressing the .u deficiencies while making the protocol address a larger subset of the functionality provided by Linux.

As stated previously, The new 9P2000.L extension will will exist in a complimentary op-code name space – that is to say operation identifiers will not overlap with existing operations, and all extensions will use new operations as opposed to changes to existing operations. Alternate extension op-codes spaces may be negotiated by optional postfixes during version negotiation – but every effort will be made to keep operations within the existing name space. It was also decided that all extensions should be treated as optional, servers which donť wish to implement them (or any subset of them) simply returns error – well behaved clients will fall back to the core 9P2000 operations.

3.4 KVM and QEMU

KVM is a set of Linux kernel modules that allows a userspace process to execute code in a special process mode. On x86, this is often referred to as compressed ring 0. Much like vm86 syscall, this mode allows a userspace program to trap interesting events such as I/O operations.

QEMU uses the interfaces provided by KVM to implement full system virtualization emulating standard PC hardware such as an IDE disk, VGA graphics, PCI networking, etc. In the case of KVM, any I/O requests a guest operating system makes are intercepted and routed to the user mode to be emulated by the QEMU process [7].

3.5 VirtIO Transport

VirtIO is a paravirtual IO bus based on a hypervisor neutral DMA API. With KVM on x86, which is the dominant platform targetted by this work, the underlying transport layer is implemented in terms of a PCI device.

A PCI device is used to enable support for the widest variety of guests since all modern x86 operating systems support PCI. The VirtIO PCI implementation makes extensive use of shared memory. This includes the use of lockless ring queues to establish a message passing interface and indirect reference to scatter/gather buffers to enable zero-copy bulk data transfer.

These properties of the VirtIO PCI transport allow VirtFS to be implemented in such a way that guest driven I/O operations can be zero-copy. This is a key advantage of VirtFS compared to using a network file system which would always require at least one (but usually more) data copies.

4 Implementation

4.1 Details of the Implementation of VirtFS server in QEMU

Making the VirtFS server part of QEMU is a very natural design decision. KVM+QEMU+VirtIO presents an ideal platform for the VirtFS server where it can efficiently interact with the guest providing one of the efficient paravirtual network file system interfaces. VirtFS server is facilitate in QEMU by defining two types of devices.

One is virtio-9p-pci device, and this will be used to transport protocol messages and data between the host and the guest. Second one is fsdev device, this is used to define the export file system characteristics like file system type, security model (section 4.2)

A typical usage of QEMU to export a file system is:

```
-fsdev local,id=exp1,path=/tmp/,security_model=mapped
-device virtio-9p-pci,fsdev=exp1,mount_tag=v_tmp
```

On the client it can be mounted with:

```
$ mount -t 9p -o trans=virtio v_tmp /mnt
```

4.2 Security Model

Security is one of the most important aspects of the file system design and it needs to be handled with care to cater specific needs of the exploiters. There are two major use cases that can make use of this new technology and their security requirements are quite different. One demands a complete isolation of guest user domain from that of the hosts hence practically eliminating any security issues related to setuid/setgid and root. This is a very practical use case for certain classes of cloud workloads where multi-tenancy is a requirement. A complete isolation of user domain can practically make two competing customers share the same file system.

The other use case is playing along with the traditional network file-serving methodologies like NFS and CIFS by sharing user domains of the host and guest. This method edges out by offering flexibility to export the same file system through other network file systems along with VirtFS.

Linux being POSIX complaint, offers a simple yet powerful file system permission model: Every file system object is associated with three sets of permissions that define access for the owner, the owning group, and for others.

Each set may contain Read (r), Write (w), and Execute (x) permissions. This scheme is implemented using only nine bits for each object. In addition to these nine bits, the Set User Id, Set Group Id, and Sticky bits are used for number of special cases.

Although this traditional model is sufficient for most of the use cases, it falls short of satisfying the needs of the modern world. The need for more granular permissions eventually resulted in a number of Access Control List (ACL) implementations on UNIX like Posix ACLs, Rich ACLs, NFSv4 ACLs etc. These ACL models were developed for specific needs and has very limited degree of commonality. This poses a major challenge for network file systems as it may have to support wide variety of file systems with different ACL models [5].

VirtFS, being one of the first file systems in the genre of paravirtual file systems need to consider all options and use cases. This type of file systems need to play a dual role, where it should be a viable alternative to net work file systems like NFS and CIFS, and also it needs to fill-in the new space where it should provide special optimizations and considerations for the needs of guest operating systems. To address these special needs and the use cases explained above, we came up with two types of security models for VirtFS: the mapped security model and the passthrough security model. QEMU administrator picks a model at the start-up and is expected to stick with that.

4.2.1 Security model: mapped

In this security model, VirtFS server intercepts and maps the file object create and get/set attribute requests. Files on the fileserver will be created with VirtFS server's (QEMU) user credentials and the client-user's credentials are stored in extended attributes. On the request to get attributes, server extracts the client-user's credentials from extended attributes and sends them to the client. Since the files are created on the fileserver with QEMU credentials, this model keeps the guests user domain completely isolated from the host's user domain. Host view shows QEMU as the owner of all files created by any user(including root) on the guest hence provides complete isolation and security.

The access permissions for user attributes are defined by the file permission bits. The file permission bits of regular files and directories are interpreted differently from the file permission bits of special files and symbolic links. For regular files and directories the file permission bits define access to the file's contents, while for device special files they define access to the device described by the special file. The file permissions of symbolic links are not used in access checks. These differences would allow users to consume file system resources in a way not controllable by disk quotas for group or world writable special files and directories. For this reason, extended user attributes are only allowed for regular files and directories only.

Given that the user space extended attributes are available to regular files only, special files are created as regular files on the fileserver and appropriate mode bits are added to the extended attributes. This method presents all special files and symlinks as regular files on the fileserver while they are represented as special files on the guest mount.

```
On Host:
# ls -l
drwx------. 2 virfsuid virtfsgid 4096 2010-05-11 09:19 adir
-rw-------. 1 virfsuid virtfsgid    0 2010-05-11 09:36 afifo
-rw-------. 2 virfsuid virtfsgid    0 2010-05-11 09:19 afile
-rw-------. 2 virfsuid virtfsgid    0 2010-05-11 09:19 alink
-rw-------. 1 virfsuid virtfsgid    0 2010-05-11 09:57 asocket1
-rw-------. 1 virfsuid virtfsgid    0 2010-05-11 09:32 blkdev
-rw-------. 1 virfsuid virtfsgid    0 2010-05-11 09:33 chardev

On Guest:
# ls -l
drwxr-xr-x 2 guestuser guestuser  4096 2010-05-11 12:19 adir
prw-r--r-- 1 guestuser guestuser     0 2010-05-11 12:36 afifo
-rw-r--r-- 2 guestuser guestuser     0 2010-05-11 12:19 afile
-rw-r--r-- 2 guestuser guestuser     0 2010-05-11 12:19 alink
srwxr-xr-x 1 guestuser guestuser     0 2010-05-11 12:57 asocket1
brw-r--r-- 1 guestuser guestuser  0, 0 2010-05-11 12:32 blkdev
crw-r--r-- 1 guestuser guestuser  4, 5 2010-05-11 12:33 chardev
```

Most of the file systems offer only one block for extended attributes. This limitation curbs the use of extended attributes to stored the target link location. Under this model, target link location is store as file data using write() and readlink reads it back through read()

```
On Guest:
# ls -l asymlink
lrwxrwxrwx 1 root root   6 2010-05-11 12:20 asymlink -> afile

On Host:
# ls -l asymlink
-rw-------. 1 root root    6 2010-05-11 09:20 asymlink
# cat asymlink
afile
#
```

Just like any security model, this has its own advantages and limitations. One of the main strength and weakness of this model is, the host file system will be VirtFSized. While the guest doesn't see any difference, host users and tools need to understand the security model to use the file system credentials on the host. This is a strength because it completely isolates the guest users address space, it allows the server to run as a non-privileged user, hence it involves no issues of root-squashing or setuid issues. Hence this security model makes it perfect for the guest to run in its own security island.

4.2.2 Security model: Passthrough.

In this security model, VirtFS server passes down all requests to the underlying file system. File system objects on the fileserver will be created with client-user's credentials. This can be done by setting setuid()/setgid() during creation or chmod/chown immediately after creation. At the end of create protocol request, files on the fileserver will be owned by client-user's credentials.

```
On Host:

# grep 611 /etc/passwd
hostuser:x:611:611::/home/hostuser:/bin/bash

# ls -l
-rwxrwxrwx. 2 hostuser hostuser 0 2010-05-12 18:14 file1
-rwxrwxrwx. 2 hostuser hostuser 0 2010-05-12 18:14 link1
srwxrwxr-x. 1 hostuser hostuser 0 2010-05-12 18:27 mysock
lrwxrwxrwx. 1 hostuser hostuser
5 2010-05-12 18:25 symlink1 -> file1

On Guest:
$ grep 611 /etc/passwd
guestuser:x:611:611::/home/guestuser:/bin/bash

$ ls -l
-rwxrwxrwx 2 guestuser guestuser 0 2010-05-12 21:14 file1
-rwxrwxrwx 2 guestuser guestuser 0 2010-05-12 21:14 link1
srwxrwxr-x 1 guestuser guestuser 0 2010-05-12 21:27 mysock
lrwxrwxrwx 1 guestuser guestuser 5 2010-05-12 21:25 symlink1 -> file1
```

This model lets the host tools understand the filessytem, and statistics collection and quotas enforcement will be easier. But this needs the server to run as a privileged user (root) and also exposes root/setuid security issues just like NFS

4.2.3 ACL Implemenation

Access Control Lists (ACLs) steps in where the traditional mode bits security model is not sufficient. ACLs allow fine grained control by the assignment of permissions to individual users and groups even if these do not correspond to the owner or the owning group. Access Control Lists are a feature of the Linux kernel and are currently supported by many common file systems and its support is crucial with the wide spread usage of file sharing among heterogeneous systems like Linux/Unix, and Windows. While ACLs are an essential part of comprehensive security scheme, the lack of universal standards often make the design complex and complicated and VirtFS is not an exception.

At the time of writing this paper, we are still at the design stage of implementing the following aspects of ACL.

A newly created file system object inherits ACLs from its parent directory. The common practice is a non-directory object inherits the parent default ACLs as its access ACLs and directory object inherits parent default ACLs as its default ACLs. To make things little more complicated, there are no standards on the inheritance algorithm and they differ for each ACL model.

In adition to the gid/uid/mode-bits, ACLs of the file system object will be checked before granting the requesting access to a file system object. This checking can be done either on the client or on the server. If the enforcement is on the server, it need to have the context of the client-user's credentials, which makes the protocol very bulky. For this reason it becomes a natural choice to have permission checks at the client.

We are leaning towards supporting at least NFSv4 level ACLs and employing client to do the ACL enforcement while the ACL inheritance is servers job. The server can choose to delegate it to the fileserver.

4.3 Current state of the project

At the time of writing this paper, the project is actively being worked on with a team of seven IBM engineers and the community is just starting to get excited. Several patches has been posted to the community mailing lists. Client side patches are being posted to the v9fs-developer@lists.sourceforge.net list and server side patches are being posted to the qemu-devel@nongnu.org list.

QEMU community blessed the project by accepting the VirtFS server feature patch set into the mainline. This is a significant milestone for the project. A patch set introducing the security model as explained above is also on the mailing list awaiting acceptance. We are also working towards allowing a more asynchronous model for the QEMU server which should boost performance significantly.

On the client side, the team has contributed dozens of patches to the 9P client of the Linux kernel. We are actively working on fixing pre-exisitng bugs and defining the 9P2000.L protocol extension. As mentioned above, the main intent and focus of this new protocol extension is to define an efficient Linux friendly protocol. In this process, we are contributing to the improvement and stabalization of the 9P client as a whole.

5 Performance

As mentioned earlier, VirtFS is intended to be the network file system specialist in the virtualization world. By the virtue of its design VirtFS is expected to yield better performance compared to its alternatives like NFS/CIFS. Though we are just getting started, lot of focus is given to the performance aspect of the implementation and the protocol design. This section covers the initial performance evaluation and comparisons with its counterparts.

5.1 Test Environment

The following test environment is used for the performance evaluation.

Host: 8 CPU 2.5GHz Intel Xeon server, 8 GB of memory, Qlogic 8Gb fiber channel controller, 24 disks JBOD. 2.6.34-rc3 kernel, mainline qemu.

Guest: 2 vCPU, 2GB memory running 2.6.34-rc3 kernel.

The following configuration is used for gathering performance numbers:

Configuration-1: Sequential read and write performance of VirtFS in comparison to NFS and CIFS. In this configuration, file system is mounted on the host and guest accesses the file system through VirtFS, NFS or CIFS.

Configuration-2: Sequential read and write performance of VirtFS in comparison to block-device performance. In this configuration, guest directly accesses the file system on the block device.

Setup

- For comparisons with blockdev, each block device is exported to guest as a block device (cache=writethrough). Filesystem is not mounted in the host and its mounted only on the guest for this testing.
- Each filesystem is stripped across 8 disks to eliminate single disk bottlenecks. 3 such filesystems are used for the performance analysis.
- Each filesystem is mounted in the host and exported (using defaults) to the guest over virtio-net.
- Filesystems are un-mounted, remounted and re exported before each read test to eliminate host level pagecache.

Commands

Used simple "dd" tests to simulate sequential read and sequential write patterns.

Write:
dd if=/dev/zero of=/mnt/fileX bs=<blocksize> count=<count>

Read:
dd if=/mnt/fileX of=/dev/null bs=<blocksize> count=<count>

blocksize - 8k, 64k, 2M.
count = number of blocks to do 8GB worth of IO.
All the tests are conducted in the guest with various IO block sizes.

5.2 VirtFS, NFS and CFS comparison

Figure-2 compares sequential read performance of VirtFS at various block sizes against NFS and CIFS. VirtFS clearly outperforms NFS and CIFS at all block sizes

Figure-3 compares sequential write performance of VirtFS at various block sizes against NFS and CIFS. Again, as expected VirtFS outperforms NFS and CIFS at all block sizes.

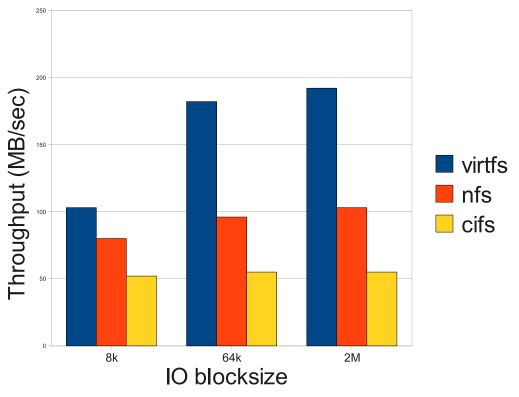

Figure 2: Comparing Sequential Read among VirtFs/NFS/CIFS

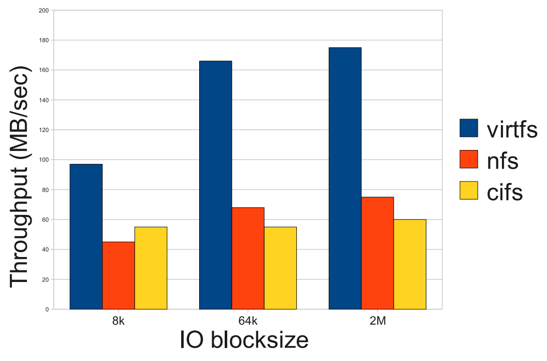

Figure 3: Comparing Sequential Write among VirtFs/NFS/CIFS

5.3 VirtFS, block device comparison

Figure-4 compares sequential read performance of VirtFS against local file system access by block device. At the time of writing this paper, VirtFS doesn't seem to scale well with number of file systems. This is due to single threaded implementation of VirtFS server in QEMU. Currently efforts are underway to convert it to multi threaded implementation.

Figure-5 compares sequential write performance of VirtFS against the block device access. As these are not synchronous writes, VirtFS is able to scale very well by taking advantage of the host and guest page caches.

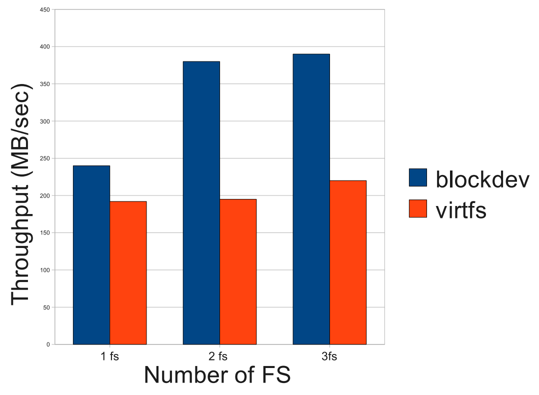

Figure 4: Comparing Sequential Read between VirtFs and Block device

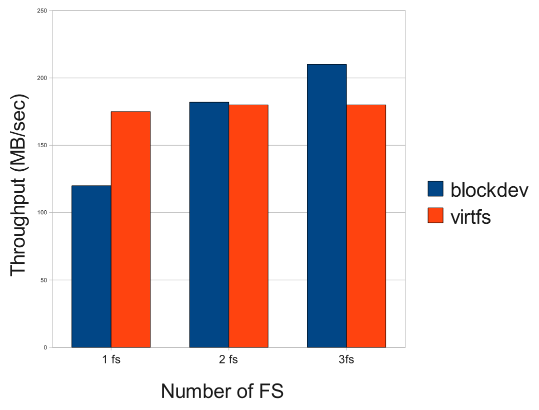

Figure 5: Comparing Sequential Write between VirtFs and Block device

6 Next Steps

6.1 Complete 9P2000.L

Defining an optimal and efficient protocol that is well suited for Linux's needs is our current focus. Implementing the defined 9P2000.L protocol messages both on the client and server and getting them into mainline will be our immediate priority. We are aiming to make the prototype available for early adopters and exploiters as quickly as possible and also plan for next releases with more advanced features.

6.2 Security

ACL implementation for network file systems is always a challenge as the server need to support different file systems with different ACL models. It becomes more complicated for VirtFS as we are supporting two different security models as mentioned above. The Linux kernel supports POSIX ACLs only but we are planning on supporting NFSv4 level ACLs. This may throw more challenges on the way.

6.3 Page Cache Sharing

Reading a file on the network file system mount on the guest makes the host to read the page onto its cache, send the data over the protocol to guests page cache before it is consumed by the user. In the case KVM, both host and guest are running on the same hardware and are using same physical resources. That means the page-cache block is being duplicated in different regions of the memory. One could extrapolate this problem to the worst case scenario where almost half of the system's page cache is wasted in duplication. We need to come up with a method where the host and guest share the same physical page. This eliminates the need for data copy between guest and host and provide better data integrity and protection to the application yielding extreme performance benefits. Sharing pages between two operating systems is challenging as we can run into various locking and distributed caching issues.

6.4 Interfacing with File system APIs

A user space server has an unique opportunity where it can interact directly with the file system API instead of going through the system call/VFS interface. VirtFS server, being a user space server can be modeled to plug directly into the fileservers API if one is available. This opens up speciality features offered by the fileserver to the guest through VirtFS and hence gives an opportunity to give a true pass-through representation of the fileserver. Simply put, this effort should provide a layer of indirection between the third party/specialized file systems on the host and virtual machines, enabling any application dependent on the special features of these file systems to run on the guests VirtFS mount.

7 Conclusions

In this paper we have motivated and described the development of a paravirtual system service, namely VirtFS. We have described its integration into KVM/QEMU and the Linux kernel, and discussed both its underlying protocol and the changes we are making to that protocol to improve its support for Linux VFS features. We have described few different use cases and included a discussion of different security models for deploying this paravirtual file systems in cloud environments. Finally, we have shown that our initial work has superior performance to the use of conventional distributed file systems and even reasonable performance when compared to the use of a paravirtualized disk. As described in the next steps (section 6) we plan on continuing the development to improve performance, stability, security, and features. It is our belief that as virtualization continues to becomes more pervasive, paravirtual system services will play a larger role – perhaps completely overtaking paravirtual devices and device emulation.

8 Acknowledgements

We would like to thank Sripathi Kodi for his valuable review comments.

We would like to thank Jim Garlick for his contributions to 9P2000.L. We'd also like to thank Ron Minnich and Latchesar Ionkov who have provided valuable feedback as well as being th eco-maintainers of the 9P subsystem in Linux. We'd also like to thank all those members of the Linux kernel development community who have contributed patches to the 9p base infrastructure we are leveraging.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[2] Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, Eric Van Hensbergen, and Robert W. Wisniewski. Libra: a library operating system for a jvm in a virtualized execution environment. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 44–54, New York, NY, USA, 2007. ACM.

[3] Bell-Labs. Introduction to the 9p protocol. *Plan 9 Programmers Manual*, 3, 2000.

[4] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM.

[5] Andreas Grunbacher. Posix access control lists on linux. *USENIX paper - Freenix track*, 2003.

[6] Eric Van Hensbergen and Ron Minnich. Grave robbers from outer space using 9p2000 under linux. In *In Freenix Annual Conference*, pages 83–94, 2005.

[7] M. Tim Jones. Discover the linux kernel virtual machine - learn the kvm architecture and advantages.

[8] Anthony Liguori and Eric Van Hensbergen. Experiences with content addressable storage and virtual disks. In *In Proceedings of the Workshop on I/O Virtualization (WIOV)*, 2008.

[9] Edmund B. Nightingale, Chris Hawblitzel, Orion Hodson, Galen Hunt, and Ross Mcilroy. Helios: Heterogeneous multiprocessing with satellite kernels. In *In Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[10] Fabio Oliveira, Gorka Guardiola, Jay A. Patel, and Eric Van Hensbergen. Blutopia: Stackable storage for cluster management. In *CLUSTER*, pages 293–302, 2007.

[11] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[12] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. *SIGOPS Oper. Syst. Rev.*, 27(2):72–76, 1993.

[13] Rusty Russel. virtio: towardsa a de-facto standard for virtual I/O devices. In *Operating Systems Review*, 2008.

[14] Adrian Schupbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[15] Eric Van Hensbergen. P.R.O.S.E.: partitioned reliable operating system environment. *SIGOPS Operating Systems Review*, 40(2):12–15, 2006.

Proceedings of the
Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada