Carnegie Mellon University
Research Showcase

Parallel Data Laboratory | Research Centers and Institutes

5-1-2008

File System Virtual Appliances: Third-party File System Implementations without the Pain (CMU-PDL-08-106)

Michael Abd-El-Malek
Carnegie Mellon University

Matthew Wachs
Carnegie Mellon University

James Cipar
Carnegie Mellon University

Gregory R. Ganger
Carnegie Mellon University

Garth A. Gibson
Carnegie Mellon University

See next page for additional authors

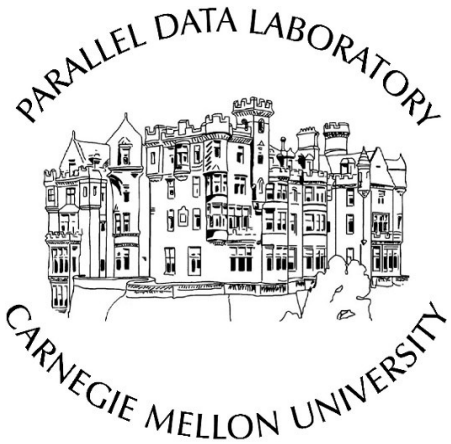
Follow this and additional works at: http://repository.cmu.edu/pdl

Recommended Citation
Abd-El-Malek, Michael; Wachs, Matthew; Cipar, James; Ganger, Gregory R.; Gibson, Garth A.; and Reiter, Michael K., "File System Virtual Appliances: Third-party File System Implementations without the Pain (CMU-PDL-08-106)" (2008). *Parallel Data Laboratory.* Paper 21.
http://repository.cmu.edu/pdl/21

This Technical Report is brought to you for free and open access by the Research Centers and Institutes at Research Showcase. It has been accepted for inclusion in Parallel Data Laboratory by an authorized administrator of Research Showcase. For more information, please contact research-showcase@andrew.cmu.edu.

Authors
Michael Abd-El-Malek, Matthew Wachs, James Cipar, Gregory R. Ganger, Garth A. Gibson, and Michael K. Reiter

This technical report is available at Research Showcase: http://repository.cmu.edu/pdl/21



File system virtual appliances: Third-party file system implementations without the pain

Michael Abd-El-Malek[1], Matthew Wachs[1], James Cipar[1],
Gregory R. Ganger[1], Garth A. Gibson[1 2], Michael K. Reiter[3]
[1] *Carnegie Mellon University*
[2] *Panasas, Inc.*
[3] *University of North Carolina at Chapel Hill*

CMU-PDL-08-106

May 2008

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

File system virtual appliances (FSVAs) address a major headache faced by third-party FS developers: OS version compatibility. By packaging their FS implementation in a VM, separate from the VM that runs user applications, they can avoid the need to provide an FS port for every kernel version and OS distribution. A small FS-agnostic proxy, maintained by the core OS developers, connects the FSVA to whatever kernel version the user chooses. Evaluation of prototype FSVA support in Linux, using Xen as the VM platform, demonstrates that this separation can be efficient and maintain desired OS and virtualization features. Using three existing file systems and a cooperative caching extension as a case study, we demonstrate that the FSVA architecture can insulate FS implementations from user OS differences that would otherwise require explicit porting changes.

Acknowledgements: We thank the members and companies of the CyLab Corporate Partners and the PDL Consortium (including APC, Cisco, EMC, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, Network Appliance, Oracle, Seagate, Symantec, and VMware) for their interest, insights, feedback, and support. This material is based on research sponsored in part by the National Science Foundation, via grants CNS-0326453 and CCF-0621499, by the Department of Energy, under Award Number DE-FC02-06ER25767, and by the Army Research Office, under agreement number DAAD19–02–1–0389. Matthew Wachs was supported in part by an NDSEG Fellowship, which is sponsored by the Department of Defense. We thank Intel and Network Appliance for hardware donations that enabled this work.

Keywords: Third-party file systems, virtual machines, extensibility

1 Introduction

Building and maintaining third-party file systems (FSs) is painful. Of course, OS functionality is notoriously difficult to develop and debug, and FSs are more so than most because of their size and interactions with other OS components (e.g., the virtual memory system). But, for third-party FSs, which are FSs not explicitly maintained by the OS implementers as a core part of the OS, there is a rarely-appreciated challenge: dealing with changes from one OS version to the next.

One would like to believe that the virtual file system (VFS) layer [20] present in most OSs insulates the FS implementation from the rest of the kernel, but the reality is far from this ideal. Instead, even when the VFS interfaces remain constant, internal FS compatibility rarely exists between one kernel version and the next. Changes in syntax, locking semantics, memory management, and preemption practices create differences that require version-specific code in the FS implementation. For "native" FSs supported by the kernel implementers (e.g., ext2 and NFS in Linux), appropriate corrections are made in the FS as a part of the new kernel version. For third-party FSs, however, they are not. As each new version is released, whether as a patch or a complete replacement, the third-party FS maintainers must figure out what changed, modify their code accordingly, and provide the new FS version. Because users of the third-party FS may be using any of the previously supported OS versions, all must be maintained and the code becomes riddled with version-specific "#ifdef"s, making it increasingly difficult to understand and modify correctly.

The pain and effort involved with third-party FSs create a large barrier for those seeking to innovate, and wear on those who choose to do so. Most researchers sidestep these issues by prototyping in just one OS version. Many also avoid kernel programming by using user-level FS implementations, via NFS-over-loopback (e.g., [5, 6, 25]) or a mechanism like FUSE (e.g., [4, 39, 15]), and some argue that such an approach sidesteps version compatibility issues. But, it really doesn't. First, performance and semantic limitations prevent most production FSs from relying on user-level approaches. Second, and more fundamentally, user-level approaches do not insulate an FS from application-level differences among OS distributions (e.g., shared library availability and file locations) or from kernel-level issues (e.g., handling of memory pressure). So, third-party FS developers address the problem with brute force.

This paper promotes a new approach (see Figure 1) for third-party FSs, leveraging virtual machines to decouple the OS version in which the FS runs from the OS version used by the user's applications. The third-party FS is distributed as a *file system virtual appliance* (FSVA), a pre-packaged virtual machine (VM) loaded with the FS. The FSVA runs the FS developers' preferred OS version, with which they have performed extensive testing and tuning. The user(s) run their applications in a separate VM, using their preferred OS version. Because it runs in a distinct VM, the third-party FS can be used by users who choose OS versions to which it is never ported.

For this FSVA approach to work, an FS-agnostic proxy must be a "native" part of the OS—it must be maintained across versions by the OS implementers. The hope is that, because of its small size and value to a broad range of third-party FS implementers, the OS implementers would be willing to adopt such a proxy. The integration and maintenance of FUSE, a kernel proxy for user-level FS implementations, in Linux, NetBSD, and OpenSolaris bolster this hope.

This paper details the design and implementation of FSVA support in Linux, using Xen as the VM platform. The Xen communication primitives allow for reasonable performance, relative to a native in-kernel FS implementation—for example, an OpenSSH build runs within 15% of the native performance. Careful design is needed, however, to ensure FS semantics, maintain OS features like a unified buffer cache, minimize OS changes in support of the proxy, and avoid loss of virtualization features such as isolation, resource accounting, and migration. Our prototype system realizes all of these design goals.

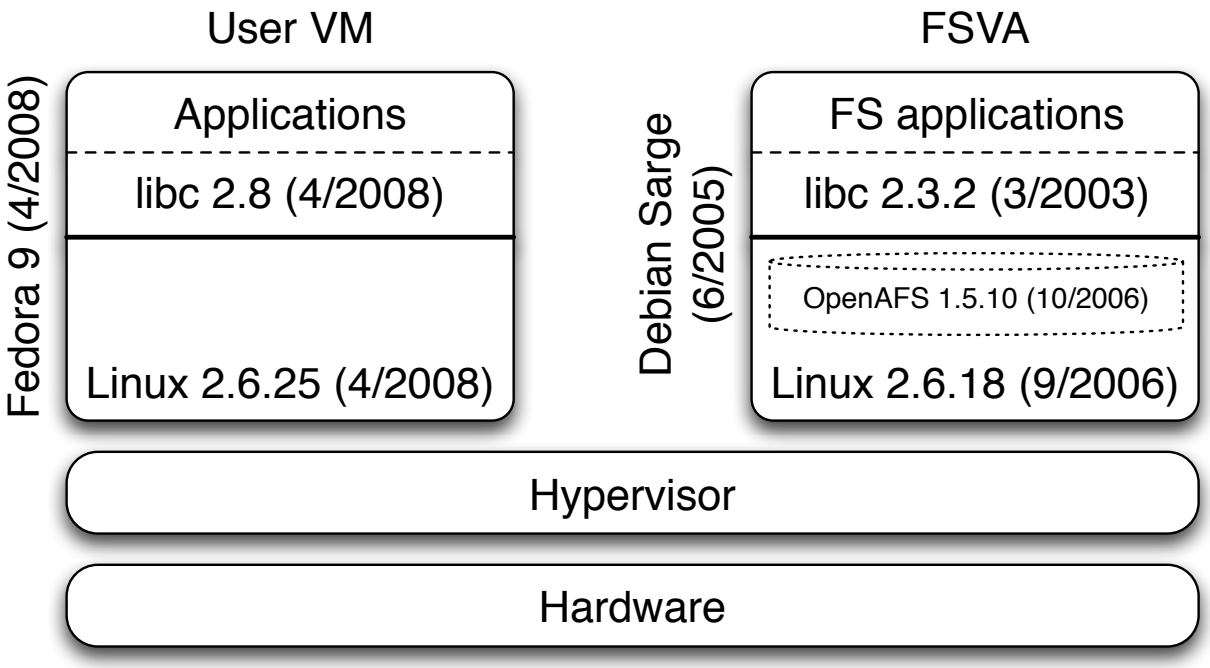


Figure 1. A file system runs in a VM provided by the third-party FS vendor. A user continues to run their preferred OS environment (including kernel version, default distribution, and library versions). By decoupling the user and FS OSs, one allows users to use any OS environment without needing a corresponding FS port from the FS vendor. This illustrated example is used as a case study in §6.2.

The efficacy of the FSVA architecture is demonstrated with a number of case studies. Three real file systems (OpenAFS, ext2, and NFS) are transparently provided, via an FSVA, to applications running on a different VM, which can be running a different OS version (e.g., Linux kernel versions 2.6.18 vs. 2.6.25). No changes were required to the FS implementation in the FSVA. In contrast, analysis of the change logs for these file systems shows that significant developer effort was required to make them compatible, in the traditional approach. To further illustrate the extensibility enabled by the FSVA architecture, we demonstrate an FS specialization targeted for VM environments: intra-machine cooperative caching eliminates redundant I/O from district VMs that share files in a distributed FS. This extension can be supported in an FSVA with no modification to the user OS. Note that, with the FSVA architecture, resources previously applied to version compatibility can instead go toward such feature addition and tuning.

2 Supporting third-party FSs

A third-party FS is developed and maintained independently of the OS with which it is intended to be used. It links into interfaces provided by that OS, after the fact. This section discusses the inter-version compatibility issues that arise from this "third party" approach and how they can be mitigated.

2.1 OS version compatibility

Based on customer demand, third-party FS vendors must port their FS to different OS versions, some of which include changes to the kernel's internal interfaces. For example, the Linux kernel's internal interfaces often undergo significant changes across minor releases. In addition to kernel-level changes, user-space environments often differ significantly among Linux distributions. For example, there is variation in the pre-installed programs and libraries, file locations, and daemon configuration syntax and locations. While this problem is most pronounced for Linux, owing to its independent and decentralized development process, changes in OS interfaces pose challenges for FS vendors whenever they occur.

In-kernel FS implementations are tightly coupled to internal kernel interfaces. FSs must adhere to the VFS interface, in order to present a unified interface to applications and share common caches [20]. In addition to the VFS interface, FS implementations depend on internal memory allocation, threading, networking (for distributed FSs), and device access (for local FSs) interfaces and semantics. To support memory mapped file I/O and a unified buffer cache, FSs are also closely coupled to the virtual memory subsystem [16]. The result of all these "extra" dependencies is that, even if the VFS interface were held constant (which it is not), FS implementations would still require updates across kernel versions.

To further shed light on these difficulties, we interviewed developers of four third-party file systems: GPFS [33], OpenAFS (an open-source implementation of AFS [18]), Panasas DirectFLOW [40], and PVFS [7]. All four FSs have been widely deployed for many years. We report their developers' first-hand experiences with maintaining the Linux client-side FS code.

Interface syntax changes. The first changes that an FS developer encounters in a new OS update are interface syntax changes, due to compilation errors. The following is a representative list. Many examples were conveyed to us by these developers, and we gleaned others from looking at OpenAFS and PVFSs' source control management systems' logs. Some examples, with the corresponding Linux kernel version in parens, include:

Callbacks: the vector I/O `readv, writev` VFS callbacks were replaced with the asynchronous I/O `aio_read, aio_write` callbacks (2.6.19). `sendfile` was replaced by `splice` (2.6.23).

Virtual memory: the virtual memory page fault handlers, which can be overridden by a file system, changed interfaces (2.6.23).

Header files: `config.h` was removed (2.6.19).

Caching: the kernel cache structure constructors' and destructors' parameters changed (2.6.20).

Structures: the per-inode `blksize` field was removed (2.6.19). The process task structure no longer contains the thread pointer (2.6.22).

While some of these changes may seem trivial, they are time-consuming and riddle source code with version-specific `#ifdefs` that complicate code understanding and maintenance. Furthermore, *every* third-party FS team must deal with each problem as it occurs. Examination of the open-source OpenAFS and PVFS change logs shows both FSs contain fixes for these (and many other similar) issues. We also encountered these issues while porting our proxy.

Policy and semantic changes. Even if interfaces remain constant across OS releases, implementation differences can have subtle effects on FSs that are hard to debug. The following examples illustrate this:

Memory pressure: many RedHat Enterprise Linux 3 kernels are not very robust in low memory situations. In particular, the kernels' policy on blocking during allocation is different from other Linux kernels. RedHat acknowledged the semantic mismatch but did not fix the issue [31].

An FS vendor was forced to work around the bug by carefully controlling the number of dirty pages (via per-kernel-version parameters) and I/O sizes to the data server (thereby negatively impacting server scalability).

Write-back: Linux uses a write-back control data structure (WBCDS) to identify dirty pages that should be synced to storage. An FS fills out this data structure and passes it to the generic Linux VFS code. Linux 2.6.18 changed the handling of a sparsely-initialized WBCDS, such that only a single page of a specified page range was actually synced. This caused the FS to mistakenly assume that all pages had been synced, resulting in data corruption.

Stack size: RedHat distributions often use a smaller kernel stack size. To avoid stack overflow, once this was discovered, an FS implementation used continuations to pass request state across server threads. Using continuations has been cumbersome for their developers and complicates debugging. This example demonstrates how one supported platform's idiosyncrasies can complicate the rest of the FS.

Locking: existing inode attribute fields required the inode lock to be held during access, whereas previously no locking was required.

Radix tree: the kernel provides a radix tree library. A Linux kernel update required the least significant bit of stored values be 0 (2.6.20), breaking an FS that was storing arbitrary integers.

Because the above changes were not documented, each third-party FS team had to discover them by code analysis and kernel debugging, and then work around them.

Overall statistics. To appreciate the magnitude of the problem, consider the following statistics. Panasas's Linux portability layer supports over 300 configurations[1]. PVFS developers estimate that 50% of their maintenance effort is spent dealing with Linux kernel issues. The most frequently revised file in the OpenAFS client source code is the Linux VFS-interfacing file. An OpenAFS developer estimates that 40% of Linux kernel updates necessitate an updated OpenAFS release.

2.2 Existing approaches and shortcomings

Most OS vendors maintain binary compatibility for user-level applications across OS releases. As a result, user-level FSs have been proposed as a vehicle for portable FS implementations [4, 25, 39]. This is done either through a small kernel module that reflects FS calls into user-space [4, 39, 15] or through a loopback NFS server that leverages existing kernel NFS client support [5, 6, 25].

Unfortunately, user-level FSs are not the answer, for several reasons. First, and foremost, user-level file systems only mitigate some OS version compatibility issues. They provide no assistance with user-space differences, such as shared library availability and file locations. Some kernel level issues, most notably those regarding memory management (e.g., the RedHat memory pressure example above), can result in correctness and/or performance problems.

User-level FSs also face performance and semantic problems. They generally require many more context switches and data copies than an in-kernel FS implementation. They can also deadlock under low-memory situations [25]. Such deadlocks can be avoided by using a purely event-driven structure, as the SFS toolkit [25] does, but at the cost of restricting implementer flexibility. When using interfaces not explicitly designed for user-level FSs, such as the NFS loopback approach, user-level FS semantics are limited by the information (no `close` calls) and control (NFS's weak cache consistency) available to them.

[1] Due to differences among distributions and processor types, Panasas clusters Linux platforms by a tuple of <distribution name, distribution version, processor architecture>. Currently, Panasas supports 45 Linux platforms. In addition, within each platform, Panasas has a separate port for each kernel version. The result is that Panasas supports over 300 configurations.

FiST [41] provides an alternative approach to a portable file system implementation, using a language-based approach. FiST provides a specialized language for file system developers. The FiST compiler generates OS-specific file system kernel modules. So, given detailed information about all relevant in-kernel interfaces, updated for each OS version, FiST could address inter-version syntax changes. But, FiST would be unlikely to offer assistance with policy and semantic changes. Also, FS developers are unlikely to adopt a specialized language unless it were expressive enough to address all desirable control, which is far from a solved problem.

2.3 FSVAs = VM-level FSs

The FSVA approach promoted in this paper is similar in spirit to user-level FSs. As before, a small FS-independent proxy is maintained in the kernel. But, instead of a user-level process, the proxy allows the FS to be implemented in a dedicated VM. This approach addresses the compatibility challenges discussed above and leverages increasing adoption of virtualization for the desktop (simplifying software maintenance [32]), enterprise (enabling server consolidation, migration, and sandboxing [13, 26]), and high performance computing environments (for increasing productivity [19, 27]).

Figure 2 illustrates the FSVA architecture[2]. User applications run in a user's preferred OS environment (i.e., OS distribution and kernel version).[3] An FS implementation executes in a VM running the FS vendor's preferred OS environment. In the user OS, an FS-independent proxy registers as an FS with the VFS layer. The user proxy transports all VFS calls to a proxy in the FSVA that sends the VFS calls to the actual FS implementation. The two proxies perform translation to/from a common VFS interface and cooperate to maintain OS and VM features such as a unified buffer cache (§4.4) and migration (§4.6), respectively.

Using an FSVA, a third-party FS developer can tune and debug their implementation to a single OS version without concern for the user's particular OS version. The FS will be insulated from both user-space and in-kernel differences in user OS versions, because it interacts with just the one FSVA OS version. Even issues like the poor handling of memory pressure in RHEL kernels can be addressed by simply not using such a kernel in the FSVA—the FS implementer can choose an OS version that does not have ill-chosen policies, rather than being forced to work with them because of a user's OS choice.

3 Additional related work

File systems and VMs. Several research projects have explored running a file system in another VM, for a variety of reasons. POFS [30] provides a higher-level file system interface to a VM, instead of a device-like block interface, in order to gain sharing, security, modularity, and extensibility benefits. XenFS [24] provides a shared cache between VMs and shares a single copy-on-write file system image among VMs. Our FSVA architecture adapts these ideas to address the cross-version compatibility problems of third-party FSs. The differing goals lead to many design differences. For example, we pass all VFS calls to the FSVA to remain FS-agnostic, whereas they try to handle many calls in the user OS to improve performance. We use separate FSVAs for each user VM to maintain virtualization features, such as migration and resource accounting, whereas they focus on

[2]Our pictures illustrate "native" virtualization, in that OSs execute in VMs atop a hypervisor. The FSVA architecture can also be used with "hosted" virtualization, where the FSVA runs in a VM hosted by the user OS.

[3]The FS executing in an FSVA may be the client component of a distributed FS. To avoid client/server ambiguities, we use the terms *user* and *FSVA* to refer to the FS user and VM executing the FS implementation, respectively.

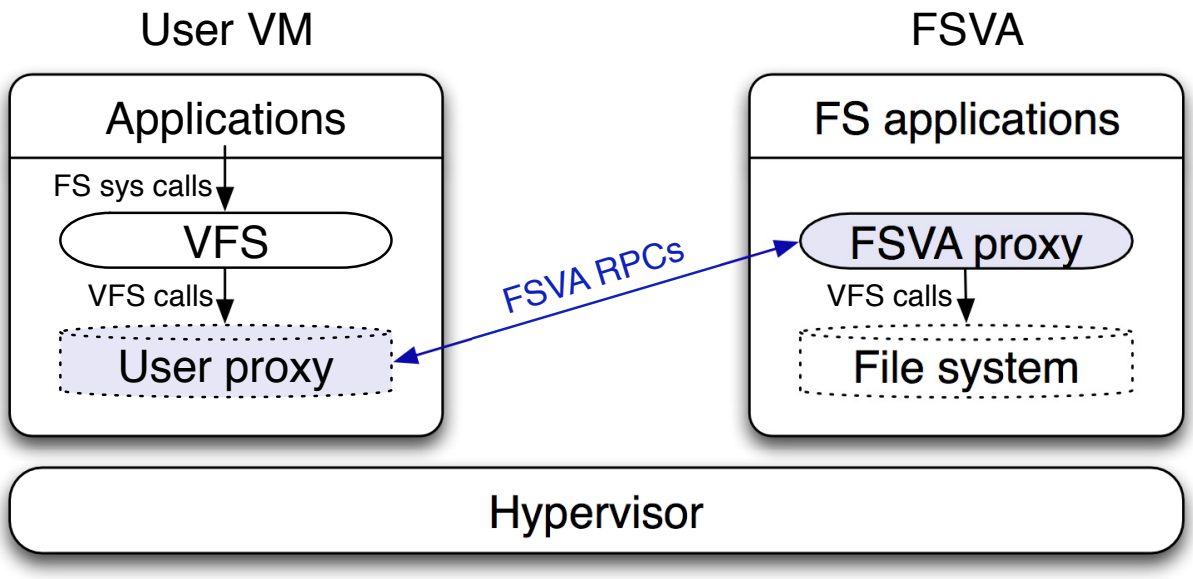


Figure 2. FSVA architecture. An FS and its (optional) management applications run in a dedicated VM. A FS-agnostic proxy running in the client OS and FSVA pass VFS calls via an efficient RPC transport.

using a single FS per physical machine to increase efficiency. We duplicate metadata on both sides to avoid dependencies, whereas they use shared memory to improve performance. We also focus on supporting a unified buffer cache cleanly.

VNFS [42] optimized NFS performance when a client is physically co-located with a server, using shared memory and hypervisor-provided communication. VNFS is NFS-specific and hence assumes file system cooperation at both VMs. For example, VNFS lets NFS clients directly read file attributes from an NFS server's shared memory. Most of their optimizations cannot be used in an FS-agnostic architecture like FSVA.

OS structure. The FSVA architecture is an application of microkernel concepts [1]. Microkernels execute OS components in privileged servers. Doing so allows independent development and flexibility. But, traditional microkernels require significant changes to OS structure. FSVAs leverage VMs and existing hypervisor support to avoid the upfront implementation costs that held back microkernels. Libra enables quick construction of specialty OSs by reusing existing OS components in another VM [2]. In contrast to Libra, our FSVA design seeks deep integration between the user OS and the FSVA.

LeVasseur et al. reuse existing device drivers by running them in a VM [22]. This is very similar to FSVAs' reuse of unmodified file system implementations. The challenge faced is dealing with the richer FS interface while retaining OS and virtualization features.

4 FSVA design

This section describes FSVA design choices intended to achieve the following goals:

Generality The FSVA should be as FS-agnostic as the generic aspect of the associated VFS allows it to be. It should not make any assumption about FS behavior, such as consistency semantics.

No FS changes To simplify adoption and deployment, FS vendors should not have to modify their FS to run in an FSVA.

Maintain OS features Support existing user OS features such as a unified buffer cache, security policies, and memory mapping. Applications should not be aware of the FSVA separation.

Minimal OS changes To encourage OS vendor adoption, the user and FSVA proxies should require minimal changes to the OS.

Maintain VM virtues Existing virtualization features such as migration, checkpointing, performance isolation, and resource accounting should not be adversely affected.

Efficiency FSVAs should have minimal overheads.

4.1 User OS proxy to FSVA interface

Together, the first three goals allows third-party FS developers to exploit FSVAs, knowing that the kernel-maintained proxies will work for them, without being required to change their FS or the OS within which they implement it. (They may choose to make changes, for efficiency, but can rely on unchanged FS semantics.) Achieving these goals, however, dictate characteristics of the interfaces between the component, especially between the proxies in the user OS and FSVA OS.

To illustrate the above point, consider the following design question: should the user OS proxy directly handle some VFS calls, without involving the FSVA? For example, should some mutating operations (such as `create` or `write`) be handled in the user OS and modifications asynchronously sent to the FSVA? Should some read-only operations (such as `read` or `getattr`) utilize user OS-cached data? Avoiding calls into the FSVA would lead to higher performance.

Unfortunately, embedding such functionality in the user OS proxy would decrease generality and couple the FSVA and user OSs. For example, many FSs carefully manage write-back policies to improve performance and achieve correctness—if the user OS performed write-back caching without giving control to the FSVA, it would lose this control and face issues such as the memory pressure and write-back issues described in §2.1. Such user OS proxy write-back would also break consistency protocols, like NFS, that require write-through. Similar problems arise for read caching in the user OS proxy: callback schemes would be needed for consistency, unless shared memory were used; a shared memory metadata cache would force the two OSs to use a common format for their cached metadata. Systems like POFS [30] and XenFS [24] choose to accept some of these consequences for the improved performance.

Our goals dictate a VFS-like interface between the proxies and that the user OS proxy forward calls rather than handling them. Most Unix OSs have similar VFS interfaces, both in the operation types (e.g., *open, create, write*) and state (e.g., file descriptors, inodes and directory entries). Consequently, the VFS interfaces in the two OSs will be similar and differences can be normalized by the proxies. In addition to VFS operations, the inter-proxy interface includes additional calls to support a unified buffer cache and migration of a user-FSVA VM pair. The return value of every VFS call to the FSVA includes updated inode attributes for inodes involved in the operation, as well as piggybacked unified buffer cache control data as described in §4.4.

4.2 Metadata duplication

In order to reduce OS changes, FS metadata is duplicated in the user OS and the FSVA. Consider the following example: in Linux, the program loading and core dumping code directly reads inode fields such as the file size *without* going through VFS calls. Put another way, there is no VFS call for "read file size." The result is that, to support in-kernel usage, the user OS must contain

an inode for that file. The FSVA will likely retain an inode for that same file to handle cached operation on subsequent calls. Thus, an inode copy exists in each VM.

Metadata duplication can be avoided at the cost of a large number of OS changes to wrap this and similar instances of inode access. But, practically, this would complicate the adoption of the user proxy by OS vendors. Given that inodes and directory entries are small data structures, we opted to duplicate metadata. As we describe in §4.4, data pages are not duplicated.

Functionally, the user OS's caching is a read- and write-through "functionality-supporting" cache. While read-only and mutating operations are always handled in the FSVA, the cache enables existing user OS features such as memory mapping and FS-based process execution to continue working.

For distributed FSs with cache consistency callbacks, it is possible that a user OS may contain stale metadata. For example, an open file's attributes may be updated in the FSVA through a cache consistency callback. But it is unlikely that this inconsistency will be visible to the user. OSs already call into the FS in response to application operations that require up-to-date attributes. This will cause the user proxy to call the FSVA and get the updated metadata.

4.3 One user OS per FSVA

A fundamental FSVA design decision is whether to share an FSVA among multiple user VMs. In our initial design, the sharing benefits of a single FSVA serving multiple user VMs favored a single FSVA approach. Common inter-VM FS metadata and data would be "automatically" shared, the number of any cache consistency callbacks would be reduced (e.g., for AFS), greater batching opportunities exist, and there exists potential for better CPU cache locality [21]. Indeed, POFS and XenFS use this single FS server approach [30, 24].

There is a well-known tension between sharing and isolation. A consequence is that the sharing opportunities provided by a single-FSVA design do not come for free. A single FSVA complicates a unified buffer cache (§4.4), resource accounting and isolation (§4.5), and user VM migration (§4.6). In each of the following subsections, we discuss how a shared FSVA would complicate these features. For now, we simply state that the second through fifth goals require a 1-to-n mapping from user VMs to FSVAs. We have found it simpler and more fitting with our goals to involve the FS with sharing in a FSVA-per-user architecture (see §7), rather than involve the FS and FSVA OS with migration, unified buffer cache, and resource accounting in a shared FSVA design.

4.4 Unified buffer cache

Early Unix OSs had separate caches for virtual memory pages and file system data. This had data and control disadvantages. On the data side, disk blocks were sometimes duplicated in both caches. On the control side, there was no single overall eviction policy, leading to suboptimal cache size partitioning. Unified buffer caches (UBCs) fix both problems [16, 34]. A single cache stores both virtual memory pages and FS data, avoiding data copies and enabling a single overall eviction policy.

An analogous problem to intra-OS separate caching (between an OS's page and buffer caches) exists when an FS runs in another VM: inter-OS separate caching (between two OSs' UBCs). To avoid extensive OS changes, we cannot coalesce the two OSs' caches into a single cache — the OSs may have different data structures and expect exclusive access to hardware (e.g., in order to read and set page access bits). Instead, we maintain the illusion of a single cache by using shared memory (to avoid data copies) and by coupling the two caches together (to use a single eviction policy). The user and FSVA proxies maintain this illusion transparently to the two OSs.

We avoid data duplication by using the hypervisor's shared memory facilities. The user and FSVA proxies map application and buffer cache data between the two OSs. This fixes the data-side problems of separate caching.

Providing a single overall eviction policy is more difficult because each OS has its own memory allocation needs and knowledge. On one hand, since applications execute in the user OS, the user OS allocates virtual memory pages and is aware of their access behavior (i.e., their access frequency). On the other hand, since I/O is performed in the FSVA (both in response to user requests and due to FS features such as read-ahead and write-back), the FSVA allocates FS buffer pages and is aware of their access behavior.

The semantic gap between the two caches can be fixed by informing one of the OSs of the other OS's memory allocation and access behavior. Since most of the allocations in the FSVA are in response to user requests, we chose to inform the user OS of the FSVA's memory allocation and access behavior. This also preserves the user OS's cache eviction semantics and performance behavior.

The FSVA proxy registers callbacks with the FSVA buffer cache's allocation and access routines. On every response to the user, the FSVA proxy piggybacks page allocation and access information. On receiving a page allocation message, the user proxy allocates a *ghost page* [10, 29] in its UBC and returns that page to the hypervisor, thereby balancing the memory usage among the OSs. On receiving a page access message, the user proxy calls the OS's page accessed function, in order to update the ghost page's order in the LRU lists. When the user OS later evicts this ghost page, the user proxy calls the FSVA to evict the corresponding page. The net result is a coupling of the two OSs' UBCs and a single overall cache eviction policy.

Compared to VM memory ballooning [37], our inter-VM UBC is VM-selective and proactive. While ballooning attempts to optimize a machine's overall memory usage, we only wish to couple a user VM and its FSVAs. This preserves the deployment-friendly ability of assigning a single memory limit to the user-FSVA VM pair and avoids inefficient memory allocation. Additionally, our fine-granularity knowledge of the FSVA's FS allocation enables us to proactively balance the two VMs' memory usage.

While UBCs enable a close coupling between the user and FSVA, a minimum amount of memory is set aside for the FSVA. This enables robust FS operation under low memory situations, thus avoiding the memory-pressure issue described in §2.1.

Our design choice of a single FSVA per user greatly simplified the UBC design. In a shared FSVA design, properly attributing page allocations and accesses to a specific user is complicated by concurrent requests and latent FS work, such as write-back and read-ahead. The FSVA OS and FS would require modifications to ensure proper attribution.

4.5 Resource accounting

Virtualization provides coarse-grained physical resource sharing among users. This low-level sharing avoids the performance crosstalk that plagues OS-level resource multiplexing [3]. In addition, coarse-grained physical resource sharing simplifies per-VM resource accounting, enabling flexible scheduling policies and accurate billing (e.g., in a shared data center).

When a user has one or more FSVAs, resource usage in the FSVAs should be charged to the user VM. This allows an administrator to continue setting a single coarse-grained resource policy for user VMs. Logically, the user VM and its FSVAs form a single unit for the purpose of resource accounting.

Associating only a single user per FSVA simplifies resource accounting. If multiple users share an FSVA, the hypervisor would not be able to map FSVA resource utilization to user VMs. Instead,

the FSVA would itself have to track per-user resource usage and inform the hypervisor. For shared block or network driver VMs [14], tracking per-user resource usage is viable, owing to the small number of requests types and their fairly regular costs [17]. But, FSVAs provide a much richer interface to users: there are many (VFS) operation types and an operation can have significantly varying performance costs (e.g., reads that hit or miss in cache). Latent OS work (e.g., cache pressure causing a previously written dirty page to be sent to the server) further complicates OS-level resource accounting. Thus, our design of a single user per FSVA simplifies resource accounting by leveraging the hypervisor's existing coarse-grained resource accounting mechanisms.

4.6 Migration

One appealing feature of virtualization is the ability to migrate VMs without OS or application support. In addition, *live migration* minimizes VM downtime, reducing interference to user applications [8]. If a VM relies on another VM for a driver [14], the VM's driver connection is reestablished to a driver VM in the new physical host. This is relatively simple since driver VMs are (mostly) stateless and have idempotent operations.

FSVAs complicate migration. In contrast to driver VMs, FSVAs potentially contain large state on behalf of a user VM and the FSVA interface is non-idempotent. There are three possible approaches to dealing with migration [9]. Either the user VM and its FSVA are simultaneously migrated, the user VM continues communicating with the FSVA at the original physical host, or a new FSVA is created at the new host and FS state is migrated. Our use of shared memory and desire to allow memory mapping of files preclude the second option, as conventional networking cannot efficiently support these features. The third option is also undesirable, since it requires FS cooperation in migrating internal state (such as extra metadata or cache consistency callbacks for distributed FSs) to the new FSVA; this breaks our design goal of no FS changes.

To allow unmodified FSs to run in an FSVA without prohibiting migration, we migrate an FSVA along with its user VM. This approach leverages VM migration's existing ability to transparently move VMs. Since some FS operations are non-idempotent, care must be taken to preserve exactly-once semantics. This is complicated by migration causing physical page mappings to change. The user and FSVA proxies transparently fix the page mappings and retransmit any in-flight requests and responses. Furthermore, we maintain live migration's low unavailability by synchronizing the two VMs' background transfer and suspend/resume.

Having only a single user per FSVA simplifies migration. In contrast, a shared FSVA would require FS involvement in migrating private state belonging to the specific user being migrated. Additionally, for distributed FSs with stateful servers, the server would need to support a client changing its IP. This would likely require server modifications. By migrating the FSVA, our approach leverages the existing migration feature of retaining IPs, thereby requiring no server changes.

4.7 Miscellaneous

Virtualization requirements. The above design requires two basic capabilities from a hypervisor: inter-VM shared memory and event notification. Popular hypervisors provide such mechanisms [3, 35]. The use of paravirtualization [3], software virtualization [36], or hardware-assisted virtualization is an implementation detail and does not affect the above design. While paravirtualization's requirement for OS changes may seem to set a high deployment barrier, in practice Xen paravirtualization support has been integrated into a number of commodity OSs (e.g., Linux, NetBSD, and OpenSolaris). Our architecture is also independent of the use of hosted versus native virtualization.

Security. Maintaining the user OS's security checks and policies is required in order to maintain the same applications semantics. Unix OSs perform generic permission-based access checks and security auditing in the VFS layer. Since the user proxy sits below the VFS layer, the existing VFS security checks continue to work. In the FSVA, the proxy calls directly into the FS, thereby bypassing the FSVA OS's security checks. In contrast to generic OS security checks, FS-specific security features may require extra effort, as described in §4.8.

Having a single user per FSVA avoids the security concerns inherent in a shared FSVA design. Also, any DoS-like activity (e.g., opening a large number of files) only harms the one user OS.

Locking. Similar to security checking, Unix OSs usually perform file and directory locking in the generic VFS layer. Consequently, the user proxy does not have to perform any file or directory locking. In the FSVA, the proxy must do the locking itself, since we are calling directly into the FS's VFS handler and bypassing the generic FSVA OS's VFS code.

4.8 Limitations

Despite an FS implementation executing in a different VM, the FSVA design endeavors to preserve FS behavior from the application's point of view. But there are a number of limitations with the FSVA architecture.

Performance. If an application spends most of its time performing in-cache FS operations, the inter-VM RPC cost will not be amortized, and the application may observe an appreciable performance degradation. Fortunately, macrobenchmarks in §6.2 show that real-world applications' performance do not suffer. Furthermore, as VM technology becomes more popular, we expect the inter-VM RPC overhead to decrease as hypervisor developers and CPU architects tackle this cost. Experience with the analogous microkernel IPC costs bolsters this hope [23].

Out-of-VFS-band state. The FSVA design fails to capture out-of-VFS-band FS state. For example, AFS [18] uses Kerberos authentication [28]. With Kerberos, a user runs a program to obtain credentials, which are stored under /tmp on a per-process-group basis. The AFS VFS handlers retrieve those Kerberos credentials. To preserve the applications' authentication semantics, the use of Kerberos authentication in AFS requires the credentials to be copied from the user OS to the FSVA. Since Kerberos is also used by other FSs, the user and FSVA proxies should probably be Kerberos-aware. However, the general problem of out-of-VFS-band state requires FS cooperation.

Incompatible FS semantics. A semantic mismatch exists if the user and FSVA OSs have incompatible VFS interfaces. For example, connecting a Unix FSVA to a Windows user OS brings up issues with regards to file naming, permission semantics [38], and directory notifications. Consequently, we envision a single FSVA for every "OS type." This paper focuses on an FSVA for Unix OSs, which tend to share similar VFS interfaces [20] and POSIX semantics [11]. It may be possible to create a superset interface to support both Windows and Unix users [12], but this is beyond our scope.

Memory overhead. There is a certain memory overhead to an FSVA, due to an extra OS and metadata duplication. Two factors mitigate this overhead. First, the FS vendor is likely to only use a small subset of the OS, leading to a small OS image. Second, since the FS vendor distributes a single FSVA, it is feasible for them to fine-tune the VM. Nevertheless, the FSVA architecture may not be appropriate for environments with severe memory pressure.

Type	Operations
Mount	`mount, unmount`
Metadata	`getattr, setattr, create, lookup, mkdir, rmdir, link, unlink, readdir, seek, truncate, rename, symlink, readlink`
Stateful ops	`open, release`
Data	`read, write, map_page, unmap_page`
Miscellaneous	`fsync`
UBC	`invalidate_page, evict_page`
Migration	`restore_grants`
Coop caching	`peer_read_page`

Table 1. The FSVA interface. Most of the calls correspond to VFS calls, with the exception of three RPCs that support migration and a unified buffer cache.

5 FSVA implementation

We implemented FSVA support in Linux using Xen. To demonstrate FS portability, we ported the user proxy to two different Linux kernels: 2.6.18 (released in September 2006) and 2.6.25 (released in April 2008).

An FSVA runs as an unprivileged VM. We modified the Xen management console scripts to support installing and removing connections between a user VM and a FSVA. When a connection is initiated, the user and FSVA proxies set up a shared memory region (containing a ring of requests and responses) and an event notification channel (for inter-VM signaling). This RPC layer closely resembles Xen's block and network drivers' RPC layers. Once a connection is made, a user can mount any FS already mounted in the FSVA. The user proxy registers as an FS with the user OS. During a mount operation, the mount options specify the FSVA identity and mountpoint.

Most of our code is implemented in user OS and FSVA kernel modules. But, we had to make three small changes to Linux and Xen. First, to allow applications to memory map FSVA pages, we modified the Linux page fault handler to call the user proxy when setting and removing page table entries that point to an FSVA page. Xen requires a special hypercall for setting user-space page table entries that point to another VM's pages. Second, to support a unified buffer cache, we added hooks to the kernel's buffer cache allocation and "page accessed" handlers. Third, to support migration, we modified the hypervisor to zero out page table entries that point to another VM at migration time.

Our migration-supporting RPC layer consists of ∼1200 lines of code (LOC). The user and FSVA proxies contain ∼3300 and ∼3000 LOC, respectively. As a reference point, the Linux NFSv3 client code is ∼11,000 LOC.

5.1 Inter-proxy interface

The majority of VFS operations have a simple implementation structure. The user proxy's VFS handler finds a free slot on the RPC ring, encodes the relevant inodes, directory entries, and flags in a generic format, and signals the FSVA of a pending request via an event notification. Upon receiving the notification, the FSVA decodes the request and calls the FS's VFS handler. Responses are handled in a reverse fashion. To avoid deadlocks like those described in §2.1 and §2.2, the user proxy does not perform any memory allocations in its RPC path.

Table 1 lists the interface between the user and FSVA proxies. Most of the RPCs correspond to VFS calls such as `mount`, `getattr`, and `read`. As described below, there is also an RPC to support migration and two RPCs to support a unified buffer cache.

There are two types of application I/O: ordinary read/write and memory mapped read/write. For ordinary I/O, the application provides a user-space buffer. The user proxy utilizes Xen's shared memory facility to create a set of *grants* for the application buffer—each grant covers one page. The I/O request contains a grant list. The FSVA proxy uses the grants to map the application buffer into the FSVA address space. The FS can then perform I/O to/from the buffer. After the I/O completes, the FSVA proxy unmaps the grant and sends the I/O response to the user. The user proxy then can recycle the grant.

Memory mapped I/O is more involved than ordinary I/O. For example, consider memory mapped read operations. When an application memory maps a file, the OS sets the application's memory page permissions to trigger a page fault when accessed. For reads, the OS page fault handler does two things. If the corresponding FS page is not in the page cache, the FS's `readpage` VFS handler is called. Once in the page cache, the application's page table pointer is then changed to point to the page. To work with an OS's memory mapping facilities, the user proxy's `readpage` VFS handler calls the FSVA to read a page (if necessary) and grant access to it. The user proxy then maps that page into the kernel's buffer cache. Application page table pointers are set on-demand in the page fault handler. When the application unmaps the file, the user proxy's `release_page` handler unmaps the grant and then calls the FSVA to allow it to recycle the grant.

To enable inter-operability between 32- and 64-bit OSs, we use compiler directives to ensure 32-bit RPC structure alignment. There is no need for XDR functions as the two VMs will have the same endianness. Due to idiosyncrasies in 32-bit Linux high memory implementation and Xen's shared memory facility, we can only map pages between VMs that are in the low memory region. This places a limit on the number of shared pages between VMs if one of them is 32-bit. We believe Xen can be modified to remove this limitation, but we have not done so.

5.2 Unified buffer cache

To maintain a UBC, the user proxy must be notified of page allocations and accesses in the FSVA. We added hooks to Linux so that the FSVA is informed of these events. When either event occurs, the FSVA proxy queues a notification. A list of these notifications are piggybacked to the user proxy on the next reply.

Linux allocates buffer cache pages in only one function, making it simple for us to capture allocation events. For page access events, there are two ways in which a page is marked as accessed. First, when an FS looks up a page in the page cache, the search function automatically marks the page as accessed in a kernel metadata structure. We added a hook to this function. Second, the memory controller sets the accessed bit for page table entries when their corresponding page is accessed. However, since all FSVA accesses to FS pages are through the search functions, we ignore this case. (Application access to memory mapped files will cause the user, not the FSVA, page table entries to be updated.)

Page evictions in the FSVA are usually triggered by the user OS. However, in some rare cases, the FSVA must ask the user OS to evict pages. Consider a file truncation. Ultimately, the FSVA decides which pages to invalidate. This choice is synchronously communicated to the user OS, ensuring that any memory-mapped pages are invalidated.

On machine startup, Linux allocates bookkeeping structures for every physical memory page. Since the FSVA's memory footprint can grow almost to the size of the initial user VM, we start the FSVA with this maximum memory size. This ensures that the FSVA creates the necessary bookkeeping structures for all the pages it can ever access. After the boot process completes, the FSVA proxy returns most of this memory to the hypervisor.

5.3 Migration

There are three steps to migrating a user-FSVA VM pair. First, the two VMs' memory images must be simultaneously migrated, maintaining the low unavailability of Xen's live migration. Second, given how Xen migration works, the user-FSVA RPC connection and the shared memory mappings must be reestablished. Third, in-flight requests and responses that were affected by the move must be reexecuted.

There are two migration facilities in Xen. Ordinary migration consists of saving a VM's memory image in one host and restoring that image in another. Live migration minimizes downtime via background copying of a VM's memory image, while the VM continues executing in the original host. After sufficient memory copying is performed, Xen suspends a VM, copies the remaining pages to the destination host, and resumes the VM.

We modified Xen's migration facility to simultaneously copy two VMs' memory images in parallel. To maintain live migration's low downtime, we synchronize the background transfer of the two memory images and the suspend/resume events. For example, if the FSVA's memory image is larger than the user VM, we delay the suspension of the user VM until the FSVA is ready to be suspended. This ensures that the user VM is not suspended at the destination host waiting for the full FSVA memory image to be migrated. Since the user VM is dependent on the FSVA, the user VM is suspended first and restored second.

When a VM is resumed, its connections to other VMs are broken. Thus, the user and FSVA proxies must reestablish their RPC connection and shared memory mappings. A shared memory mapping in Xen depends on the two VMs' numerical IDs and physical page addresses. After migration completes, both values are likely different. Consequently, all the shared memory mappings between the two VMs must be reestablished through the hypervisor. We use Xen's batched hypercall facility to speed up this process. A side-effect is that the FSVA proxy must maintain a list of all shared pages to facilitate this reestablishment. The user proxy performs a special `restore_grants` RPC to retrieve this list from the FSVA.

When a user VM is resumed, its applications may attempt to access an FSVA's memory mapped page whose mapping has not yet been restored. This access would cause an application segmentation fault. To avoid this, we modified the hypervisor's migration code to zero out user VM page table entries that point to another VM. Consequently, application attempts to access the page will cause an ordinary page fault into Linux, and the user proxy will block the application until the page's mapping has been reestablished.

Because the user-FSVA RPC connection is broken during migration, in-flight requests and responses must be resent. To enable retransmission, the user OS retains a copy of each request until it receives a response. To ensure exactly-once RPC semantics, unique request IDs are used and the FSVA maintains a response cache. Read operations are assumed to be idempotent and hence the response cache is small. The FSVA garbage collects a response upon receiving a request in the request ring slot corresponding to that response's original request.

6 FSVA evaluation

This section evaluates our FSVA prototype. First, it demonstrates that FSVAs allow two different Linux environments to use the same third-party FS implementation. Second, it quantifies the performance overheads of our FSVA prototype. Third, it illustrates the efficacy of our inter-VM unified buffer cache and live migration support.

6.1 Experimental setup

Except for the NFS and OpenAFS servers, all experiments are performed on quad-core 1.86 GHz Xeon E5320 machines with 6 GB of memory, a 73 GB SAS drive (for ext2 experiments), and a 1 Gb/s Broadcom NetXtreme II BCM5708 Ethernet NIC. Unless noted, all computers run 64-bit Linux kernel 2.6.18 with the latest Debian testing distribution. To get the best disk performance, VMs use a local disk partition for storage, instead of file-backed storage. The NFS server is a 2-CPU Xeon 3 GHz, 2 GB of memory, a 250 GB 7200 rpm Seagate SATA disk, and an in-kernel NFS server implementation. The OpenAFS server is a Sun Ultra 2 Enterprise machine, with 2x300 Mhz Ultra Sparc II processors, 1 GB of memory, a 100 Mb/s Ethernet NIC, an Arena Industrial II external RAID with 6x120GB Seagate SCSI drives, and Solaris 10 with kernel 120011-14.

Unless otherwise stated, the experiments compared two setups: a user VM versus a user VM connected to an FSVA. This allows us to focus on the overhead that separating the FS into a FSVA introduced. The inter-VM unified buffer cache allowed us to specify the same memory size in both experimental setups; the user-FSVA VM pair do not benefit from any extra caching. To ensure comparable CPU access, the single user VM setup was given two CPUs, while the user-FSVA VM pair were each given one CPU.

6.2 Cross-version FSs via FSVAs

The FSVA architecture is intended to address the cross-version compatibility challenges faced by third-party FS developers. We test its ability to do so with FSs running in FSVAs with very different Linux environments than those of a benefitting user VM. Here, we focus on the setup shown in Figure 1. The FSVA runs an old Debian Sarge distribution (2005), libc 2.3.2 (2003), and Linux kernel 2.6.18 (2006). In contrast, the user VM runs a bleeding-edge Linux environment: Fedora 9 (in alpha release), libc 2.8 (release candidate), and Linux 2.6.25 (released in April 2008). Both OSs are 32-bit — it was simpler to compile AFS for 32-bit Linux, and the mainline Linux kernel tree currently only supports 32-bit Xen VMs.

We used OpenAFS client version 1.5.10 (2006) as the test FS. OpenAFS is a compelling test case due to its sheer size. It is one of the most complex open-source third-party FSs available. The client kernel module consists of ∼31,000 OS-independent LOC and ∼7,000 Linux-specific LOC. It also contains a number of user-level programs for tasks such as volume management. As we noted earlier, one of the OpenAFS lead developers estimated that 40% of kernel updates necessitate a new OpenAFS release. Indeed, our user VM's kernel (2.6.25) caused a new OpenAFS release (1.5.35), and there were 25 other releases (many for OS version compatibility) between it and the one used in the FSVA.

We ported the user proxy to kernel 2.6.25. (Such porting would be unnecessary if Linux adopts the user proxy.) No porting of OpenAFS was needed. The user VM mounts and uses the FSVA's OpenAFS, despite the OS version differences. As one example, we compiled OpenSSH v4.7p1 in the user VM. Compilation exercises most of the VFS calls and heavily uses memory-mapped I/O in the configure phase. When executed on the FSVA's OS version (as a user VM) directly over the OpenAFS client, the compilation time was 91.5 s. When executed in the user VM connected to the FSVA, the run time was 105.4 s, a 15% overhead. Clearly, there is an overhead for passing every VFS operation across the FSVA, especially with Xen's relatively unoptimized (for fast RPC) communication mechanisms. But, it enables a complex FS such as OpenAFS to provide the same semantics to a user OS environment, despite not being ported from an OS version that is 1.5 years older.

Operation	Latency	Operation	Latency
Null hypercall	0.68	Send VIRQ	1.52
Thread switch	1.58	4 KB memcpy	1.73
Create grant	0.43	Destroy grant	0.33
Map grant	1.69	Unmap grant	2.10
Null RPC	9.64		
getattr (ext2)	1.37	getattr (FSVA/ext2)	15.3
readpage (ext2)	1.73	readpage (FSVA/ext2)	21.2

Table 2. FSVA microbenchmarks. Latencies are in μs.

6.3 Microbenchmarks

To understand the causes of the FSVA overhead, we used high-precision processor cycle counters to measure a number of events. Table 2 lists the results.

The send VIRQ operation refers to sending an event notification to another VM. A VM wishing to share a page with another VM performs the create and destroy grant operations (which involve no hypercalls), while the VM mapping the page performs the map and unmap grant operation (each involving one hypercall). It is interesting to note that it is more efficient to "share" a single page through two memory copies (say, over a dedicated staging area) than through the grant mechanism. However, since Xen allows batched hypercalls, the grant mechanism is faster than memory copies when sharing more than one page due to the amortized hypercall cost.

An RPC requires two event notifications and two thread switches, one each on the user OS and the FSVA. Those four operations correspond to 6 μs of the 9.64 μs null RPC latency we observed. The rest of the RPC time (3.64 μs) is taken up by locking the shared RPC ring, copying the request and response data structures onto the ring, and other miscellaneous operations.

We now compare the latencies of two representative FSVA RPCs: `getattr` and `readpage`. We measured the latencies over ext2 and over an FSVA running ext2. The FSVA `getattr` latency is equal to the latency of the underlying ext2's `getattr` plus the null RPC plus a few microseconds for encoding and decoding the inode identifier and attributes. The additional time in the FSVA `readpage`, as compared to `getattr`, is due to the grant calls.

6.4 Unified buffer cache

To demonstrate the unified buffer cache, we ran an experiment with an application alternating between FS and virtual memory activity. The total memory for the user VM and FSVA is 1 GB. As described in §5.2, both VMs are started with 1 GB of memory. Once the user and FSVA kernel modules are loaded, the FSVA returns most of its memory to Xen, thereby limiting the overall memory usage to 1 GB plus the size of the FSVA's OS (which we call the *overhead*).

Figure 3 shows the amount of memory each VM consumes. Starting with a cold cache, the application reads a 900 MB file through memory mapped I/O. This causes the FSVA's memory size to grow to 900 MB, plus its overhead. The application then allocates 800 MB of memory and touches these pages, triggering Linux's lazy memory allocation. As the allocation proceeds, the user VM evicts the clean FS pages to make room for the virtual memory pressure. These eviction decisions are sent to the FSVA; the FSVA then returns the memory to the user VM. Linux apparently evicts a a large batch of file pages initially, then trickles the remainder out.

In the third phase, the application performs a 500 MB ordinary read from a file. This requires FS pages to stage the data being read. Since the application has not freed its previous 800 MB allocation, and swapping is turned off for this experiment, the virtual memory pages cannot be

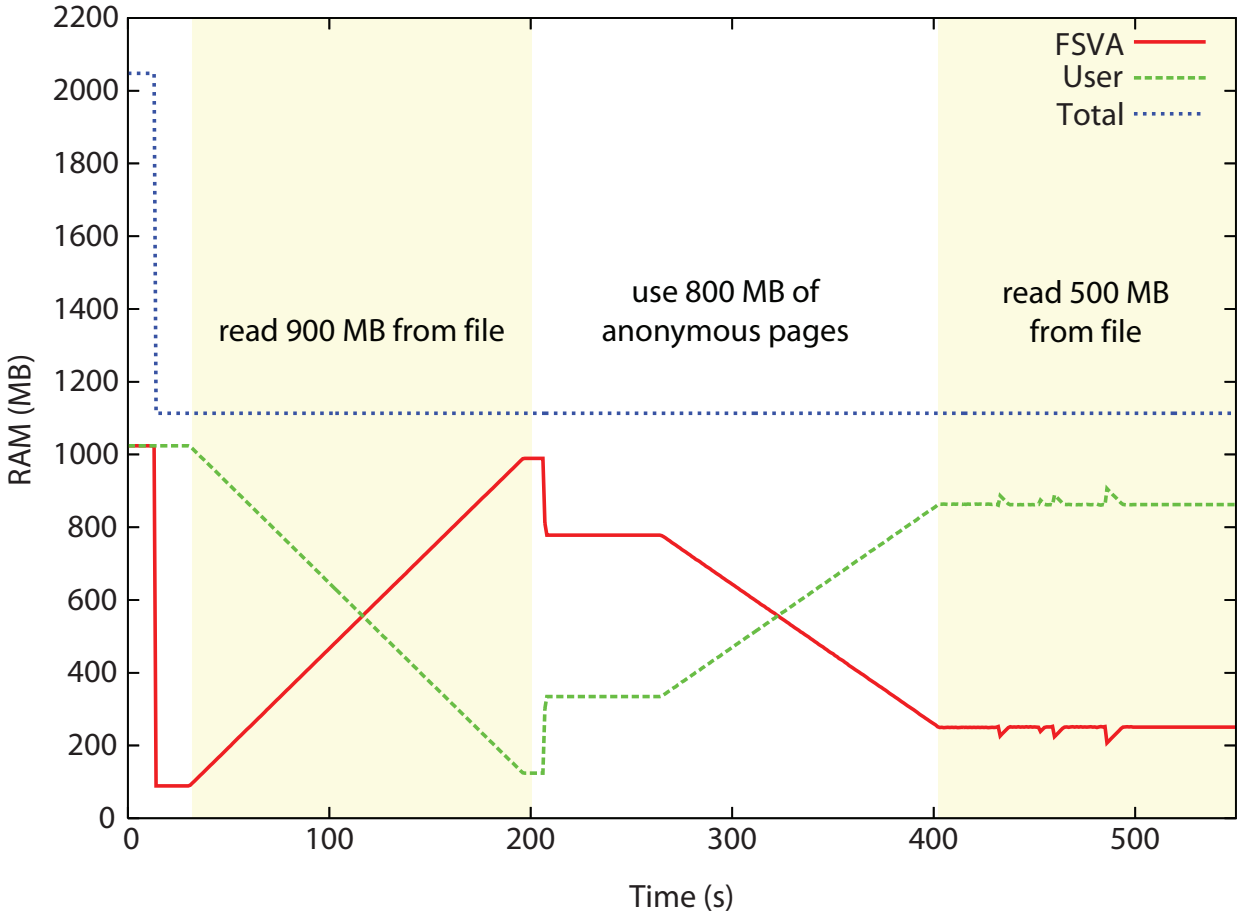


Figure 3. Unified buffer cache. This figures shows the amount of memory consumed by the user and FSVA VMs. As applications shift their memory access pattern between file system and virtual memory usage, the unified buffer cache dynamically allocates memory among the two VMs while maintaining a constant total memory allocation.

evicted. The result is that only the remaining space (just over 200 MB) can be used to stage reads; the unified buffer cache constrains the FSVA to this size. Linux sometimes batches page eviction, causing the dips in the figure.

6.5 Migration

To evaluate the FSVA's effect on unavailability during live migration, we wrote a simple benchmark that continuously performs read operations on a memory-mapped file. This allows us to measure the slowdown introduced by migrating the user-FSVA VM pair. Every microsecond, the benchmark reads one byte from a memory-mapped file and sends a UDP packet containing that byte to another machine. This second machine logs the packet receive times, providing an external observation point.

To establish baseline live migration performance, we ran our benchmark against the root NFS filesystem of a single VM with 512 MB of memory. During live migration, the unavailability period was 0.29 s. We then repeated this test against the same FS exported from an FSVA to a user VM. The two VMs' memory allocation was set to 512 MB plus the overhead of the FSVA's operating

system, which was approximately 92 MB. Unavailability increased to 0.51 s. This increase is caused by the extra OS pages that need to be copied during the suspend phase, as well as the overhead of our RPC layer and shared memory restoration. We believe this overhead is relatively independent of the overall memory size, but we were unable to run larger migration experiments due to limitations in preallocated shadow page tables that Xen uses during migration.

7 Cooperative caching

Overview. By allowing an FS vendor to focus on just one OS environment, the FSVA architecture simplifies the addition and maintenance of FS features. To demonstrate this extensibility, we describe and evaluate an intra-machine cooperative caching feature that is specifically targeted for VM environments. We chose cooperative caching in order to demonstrate how our single FSVA per user OS can make use of inter-VM sharing opportunities. While this optimization could be done without FSVAs, it becomes much easier to deploy and maintain with this architecture.

To efficiently use a machine's limited physical memory, a hypervisor can transparently deduplicate pages among VMs [37]. While this feature creates "extra" memory pages, it can only detect common pages once VMs have read them into memory. Our cooperative caching feature solves an orthogonal problem: avoiding I/O in the first place. If a VM is about to read a page from a file server, it first checks to see if other VMs already contain this page. If the page is present, the VM can simply copy the page contents and avoid the I/O.

Unlike most of the FSVA design, I/O-suppressing cooperative caching is highly FS-specific. An FS's cache consistency semantics dictate whether a client can use another client's page. For concreteness, we discuss and demonstrate a prototype of this feature with NFSv3.

Design. NFS cache consistency is based on timeouts. When an application performs a read operation, the NFS client's VFS handler checks the inode's age. If the inode has exceeded its timeout, a fresh copy of the inode is fetched from the server. If the fetched inode differs from the cached one, all of the file's pages are flushed from the cache. The timeout is set by a heuristic based on how long the inode has been remained in cache without being modified.

To preserve NFS's page-granularity access, a cooperative caching protocol for NFS must also operate on a per-page basis. Consequently, when our cooperative NFS client needs to read a page from an NFS server, it first asks a *peer VM* for a copy of that page. If the peer's cache is valid, it copies the data into a supplied buffer. If the cached copy is stale, we short-circuit the NFS client's readpage handler, preventing the peer from refreshing the page from the server. Clients only copy pages from their memory and do not perform network requests on behalf of their peers.

Implementation. To demonstrate that cooperative caching can be implemented without modifying the user OS, or even its FSVA proxy, we implemented a simple prototype of our NFS design based on the Linux kernel's NFS client. This prototype makes no attempt to perform bulk reads, or any other optimizations that may be possible for NFS cooperative caching. Because of this, it suffers a performance overhead if a large fraction of cooperative cache reads fail. However, it is sufficient to show that such features can be implemented without modifying the user OS.

Evaluation. We tested our cooperative caching implementation under a random access workload. Two user VMs were configured to mount the same NFS export via separate FSVAs. They then perform 100,000 random access operations to a 400MB memory mapped file. This test was run in three modes. First, the workloads on separate VMs were started simultaneously. Second, they were started sequentially. Third, the second workload was started 5 seconds after the first.

When the two workloads were started simultaneously, there was no benefit to cooperative caching, and the overhead of the cooperative caching RPCs caused an 11% slowdown. However,

when started at different times, either sequentially or within 5 s of each other, cooperative caching provided a speedup of over 40%. We also tested our cooperative caching system with sequential access patterns, but the overhead of page-by-page cooperative operations could not match the efficiency of bulk NFS operations. A more sophisticated cooperative caching implementation could use bulk operations to achieve similar efficiency.

8 Conclusion

FSVAs offer a solution to the cross-version compatibility problems that plague third-party FS developers. The FS can be developed, debugged, and tuned for one OS version and bundled with it in a preloaded VM (the FSVA). Users can run whatever OS version they like, in a separate VM, and use the FSVA like any other file system. Case studies show this approach works for a range of FS implementations across distinct OS versions, with minimal performance overheads and no visible semantic changes for the user OS.

Acknowledgements

We thank Chris Behanna (Panasas), Derrick Brashear (OpenAFS), Nitin Gupta (Panasas), Paul Haskin (GPFS), Sam Lang (PVFS), Rob Ross (PVFS), and Brent Welch (Panasas) for sharing their FS development experience. We thank Ben Pfaff (POFS), Mark Williamson (XenFS), and Xin Zhao (VNFS) for sharing their code. Last, but not least, we thank Adam Pennington for providing us with access to his AFS server and Michael Stroucken for installing and maintaining our servers.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. Summer USENIX Conference, pages 93–112. Usenix Association, 1986.

[2] G. Ammons, J. Appavoo, M. Butrico, D. D. Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. V. Hensbergen, and R. W. Wisniewski. Libra: a library operating system for a jvm in a virtualized execution environment. VEE, pages 44–54. ACM, 2007.

[3] 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. SOSP, pages 164–177. ACM, 2003.

[4] B. N. Bershad and C. B. Pinkerton. Watchdogs: Extending the UNIX File System. USENIX Winter Conference, pages 267–275. USENIX Association, 1988.

[5] M. Blaze. A cryptographic file system for UNIX. ACM CCS, pages 9–16. ACM, 1993.

[6] B. Callaghan and T. Lyon. The automounter. USENIX Winter Conference, pages 43–51. USENIX Association, 1989.

[7] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. Annual Linux Showcase and Conference, pages 317–327. USENIX Association, 2000.

[8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. NSDI, pages 273–286. USENIX Association, 2005.

[9] F. Douglis and J. K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, **21**(8):757-785, 1991.

[10] M. Ebling, L. Mummert, and D. Steere. Overcoming the Network Bottleneck in Mobile Computing. WMCSA, 1994.

[11] H. Eifeldt. POSIX: a developer's view of standards. USENIX ATC, pages 24–24. USENIX Association, 1997.

[12] M. Eisler, P. Corbett, M. Kazar, D. S. Nydick, and C. Wagner. Data ONTAP GX: a scalable storage cluster. FAST, pages 23–23. USENIX Association, 2007.

[13] P. Fabian, J. Palmer, J. Richardson, M. Bowman, P. Brett, R. Knauerhase, J. Sedayao, J. Vicente, C.-C. Koh, and S. Rungta. Virtualization in the Enterprise. *Intel Technology Journal*, **10**(3):227–242. Intel, August 10, 2006.

[14] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. *Reconstructing I/O*. Technical report. University of Cambridge, Computer Laboratory, 2004.

[15] FUSE. FUSE: filesystem in userspace, 2008. `http://fuse.sourceforge.net`.

[16] R. A. Gingell, J. P. Moran, and W. A. Shannon. Virtual Memory Architecture in SunOS. *USENIX Summer Conference*, pages 81–94, 1987.

[17] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. Middleware, pages 342-362, 2006.

[18] 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. *TOCS*, **6**(1):51–81. ACM, 1988.

[19] W. Huang, J. Liu, B. Abali, and D. K. Panda. A case for high performance computing with virtual machines. ICS, pages 125–134. ACM Press, 2006.

[20] S. R. Kleiman. Vnodes: an architecture for multiple file system types in Sun Unix. Summer USENIX Conference, pages 238–247. USENIX Association, 1986.

[21] J. R. Larus and M. Parkes. Using Cohort-Scheduling to Enhance Server Performance. USENIX ATC, pages 103–114. USENIX Association, 2002.

[22] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. OSDI, pages 17–30. USENIX Association, 2004.

[23] J. Liedtke. Improving IPC by kernel design. SOSP, pages 175–188. ACM Press, 1993.

[24] Mark Williamson. XenFS, 2008. `http://wiki.xensource.com/xenwiki/XenFS`.

[25] D. Mazieres. A toolkit for user-level file systems. USENIX ATC. USENIX Association, 2001.

[26] L. McLaughlin. Virtualization in the Enterprise Survey: Your Virtualized State in 2008. *CIO Magazine*. CXO Media Inc., January, 2008.

[27] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis. Virtualization for high-performance computing. *SIGOPS Oper. Syst. Rev.*, **40**(2):8–11. ACM Press, 2006.

[28] B. C. Neuman and T. Ts'o. Kerberos: an authentication service for computer networks. *IEEE Communications*, **32**(9):33–38, Sep. 1994.

[29] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. SOSP, pages 79–95. ACM Press, 1995.

[30] B. Pfaff. *Improving Virtual Hardware Interfaces*. PhD thesis. Stanford, October 2007.

[31] RedHat. Bug 111656: In 2.4.20.-20.7 memory module, rebalance_laundry_zone() does not respect gfp_mask GFP_NOFS, 2004. `https://bugzilla.redhat.com/show_bug.cgi?id=111656`.

[32] C. Sapuntzakis and M. S. Lam. Virtual appliances in the collective: a road to hassle-free computing. HotOS, pages 10–10. USENIX Association, 2003.

[33] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. FAST, page 19. USENIX Association, 2002.

[34] C. Silvers. UBC: an efficient unified I/O and memory caching subsystem for NetBSD. USENIX ATC, pages 54–54. USENIX Association, 2000.

[35] VMWare. Virtual Machine Communication Interface, 2008. `http://pubs.vmware.com/vmci-sdk/index.html`.

[36] VMWare. VMWare ESX Server Product Overview, 2008. `http://www.vmware.com/products/vi/esx/`.

[37] C. Waldspurger. Memory resource management in VMware ESX server. OSDI, pages 181–194, 2002.

[38] A. Watson, P. Benn, and A. G. Yoder. *Multiprotocol Data Access: NFS, CIFS, and HTTP*. Technical report. Network Appliance, 2001.

[39] N. Webber. Operating system support for portable filesystem extensions. USENIX Winter Conference, pages 219–228. USENIX Association, 1993.

[40] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. FAST, pages 1–17. USENIX Association, 2008.

[41] E. Zadok and J. Nieh. FiST: A language for stackable file systems. USENIX ATC, pages 55–70. USENIX Association, 2000.

[42] X. Zhao, A. Prakash, B. Noble, and K. Borders. *Improving Distributed File System Performance in Virtual Machine Environments*. CSE-TR-526-06. University of Michigan, September 2006.