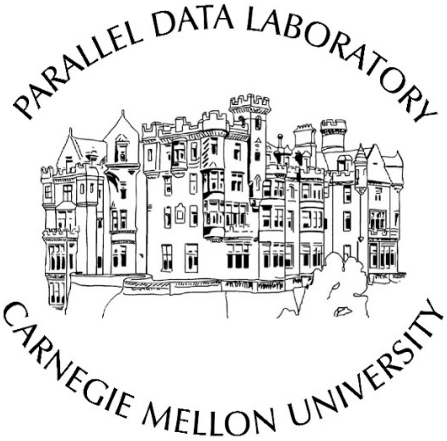

File system virtual appliances: Portable file system implementations

Michael Abd-El-Malek[1], Matthew Wachs[1], James Cipar[1], Karan Sanghi[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-10-105
April 2010

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

Abstract

File system virtual appliances (FSVAs) address the portability headaches that plague file system (FS) developers. By packaging their FS implementation in a VM, separate from the VM that runs user applications, they can avoid the need to port the file system to each OS and OS version. A small FS-agnostic proxy, maintained by the core OS developers, connects the FSVA to whatever OS the user chooses. This paper describes an FSVA design that maintains FS semantics for unmodified FS implementations and provides desired OS and virtualization features, such as a unified buffer cache and VM migration. Evaluation of prototype FSVA implementations in Linux and NetBSD, using Xen as the VMM, demonstrates that the FSVA architecture is efficient, FS-agnostic, and able to insulate file system implementations from OS differences that would otherwise require explicit porting.

Acknowledgements: We thank the members and companies of the CyLab Corporate Partners and the PDL Consortium (including APC, Data Domain, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, IBM, Intel, LSI, Microsoft Research, NetApp, Oracle, Seagate, Sun Microsystems, 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: Portable file systems, third-party file system development, operating systems, virtual machines

1 Introduction

Building and maintaining file systems (FSs) is painful. 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. In-kernel FSs must adhere to the OS's virtual file system (VFS) interface [18], but that is the easy part. FS implementations also depend on memory allocation, threading, locking/preemption, 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.

Though difficult during initial FS development, these "extra" dependencies particularly complicate porting a file system to different OSs or even OS versions. While VFS interfaces vary a bit across OSs, the other OS internal interfaces vary greatly, making porting and support of file systems painful and effort-intensive. Need for portability comes in three forms: (1) an FS developed for one OS requires explicit porting to function in another; (2) an FS developed for one OS version requires modifications to function in each subsequent version of that OS, which is particularly burdensome for third-party FS developers; (3) an FS developed for the latest OS version must be backported, to support users of a previous OS version who cannot upgrade to the latest version.

In practice, these portability issues require substantial developer effort—approximately 50% of the effort, in the estimate of some developers (see §2.1). Most researchers sidestep these issues by prototyping in just one OS (version). Many also avoid kernel programming by using user-level FS implementations, via a mechanism like FUSE (e.g., [4, 36, 14]) or NFS-over-loopback (e.g., [6]), and some argue that such an approach sidesteps version compatibility issues. In reality, it does not, for both practical and fundamental reasons. For example, there are many existing in-kernel FS implementations that would require a user-level re-implementation. User-level approaches also do not insulate an FS from OS-specific kernel-level issues (e.g., handling of memory pressure) or from user-level differences among OSs (e.g., shared library availability and file locations). So, FS developers address the problem with brute force where they can, and simply forgo OSs that pose too large a hurdle.

This paper offers a new approach (Figure 1) for portable FS implementations, leveraging virtual machines (VMs) to decouple the OS in which the FS runs from the OS used by the user's applications. The FS is distributed as a *file system virtual appliance* (FSVA), a pre-packaged virtual appliance [29] 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). Since it runs in a distinct VM, the FS can be used by users who choose OSs to which it is never ported. The FSVA approach handles all three forms of the FS portability problem.

Crucially, the FSVA approach relies on the interface *to* FSs being relatively simple and consistent across OSs, with a small-ish number of VFS primitives. In contrast, the interfaces *from* FSs to other OS components are much more complex and varying, especially when trying to maximize FS performance. The VFS-like FSVA interface allows a FS in an FSVA to be accessed via an FS-agnostic proxy in the user's VM, isolated from the user OS's internal interfaces.

For the FSVA approach to work, the 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 FS users and implementers, the OS implementers would be willing to adopt such a proxy. FUSE, a kernel proxy for user-level FS implementations, has been integrated into Linux, NetBSD, and OpenSolaris, and we envision a similar adoption path for the FSVA proxy.

This paper details the design, implementation, and evaluation of FSVA support in Linux and

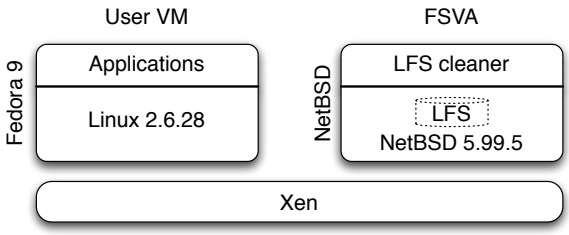

Figure 1. A file system runs in a separate VM. A user continues to run their preferred OS. By decoupling the user and FS OSs, one allows users to use any OS without needing a corresponding FS port. As an example, Linux does not include a log-structured file system (LFS) implementation. But, using FSVAs, a Linux user can utilize NetBSD's LFS implementation. This illustrated example is used as a case study in §5.2.

NetBSD, using Xen as the VM platform. The Xen communication primitives allow for reasonable performance—for a variety of macrobenchmarks and FSs, the slowdown is less than 15% compared to native in-kernel FSs with a very small FS change. 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 retain virtualization features such as isolation and migration. Our prototype realizes all of these design goals.

We demonstrate the efficacy of the FSVA architecture in addressing the FS portability problem with a number of case studies. Six FSs (ext2, ext3, ext4, LFS, NFS, and ReiserFS) are transparently provided, via an FSVA, to applications running on a different VM, which can be running a different OS or different OS version. For example, a Linux user can utilize the NetBSD log-structured FS [28] implementation, immediately filling the void of LFS implementations for Linux. As another example, a Linux 2.6.18 user can immediately use the new ext4 FS, which was recently merged into the Linux 2.6.28 kernel but is not available in 2.6.18. No changes are required to the FS implementations in the FSVA to enable such portability.

2 Porting FS implementations

FS implementations are tightly intertwined with OS internals. Of course, the OS calls into the FS to access its functions. The VFS interface [18] in most OSs allows multiple FSs to co-exist in an OS, while presenting a unified interface and sharing common caches. The VFS approach was also intended to ease portability of FSs across OSs [18, 36], but it falls far short of its goal. The problem is that the majority of portability problems relate to FS-OS interdependencies other than the basic FS interface. Specifically, to implement its functionality, the FS must rely on and conform to many internal OS interfaces and semantics, including memory allocation, threading, locking/preemption, and the virtual memory subsystem. These aspects vary widely across OSs, and they often vary even across versions of the same OS. Adapting to such variation is the primary challenge in porting FS implementations.

This section describes portability challenges in more detail, the shortcomings of existing approaches, and the FSVA approach to addressing FS portability.

2.1 Porting experiences from the field

To better understand FS portability, we interviewed developers of four third-party FSs: GPFS [30], OpenAFS, Panasas DirectFLOW [38], and PVFS [7]. All four FSs have been widely deployed

for many years. Since the inter-OS FS porting problem is well-known and PVFS and Panasas DirectFLOW are only available in Linux, we describe the four FS developers' first-hand experiences with intra-OS FS porting: maintaining Linux client-side FS code. Naturally, inter-OS portability faces all of these issues and more.

Interface syntax changes. The first changes that an FS developer encounters in an OS update are interface syntax changes, due to compilation errors. The following is a representative list. Many examples were conveyed to us by the 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 parentheses, 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, overridable by an FS, changed interfaces (2.6.23).

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 that both FSs contain fixes for these (and many similar) issues.

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

Memory Pressure: some RedHat Enterprise Linux 3 kernels are not robust in low memory situations. In particular, the kernels can block during allocation despite the allocation flags specifying no blocking. This resulted in minutes-long delays in dirty data writeback under low-memory situations. RedHat acknowledged the semantic mismatch but did not fix the issue [27]. 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 stable 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 an 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 (4 K instead of the default 8 K). To avoid stack overflow, once this was discovered, an FS implementation used continuations to pass request state across server threads. Continuations have been cumbersome for the developers and complicate debugging. This illustrates how one supported platform's idiosyncrasies can complicate the entire FS, not just the OS-specific section.

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

Overall statistics. To appreciate the magnitude of the problem, consider the following statistics. Panasas' Linux portability layer supports over 300 configurations. 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.

One may be tempted to brush off the above difficulties as artificial and related only to Linux. But, while most pronounced for Linux, with its independent and decentralized development process, this problem poses challenges for FS vendors targeting any OS. Furthermore, given Linux's popularity in the server marketplace, this is a real problem faced by third-party FS developers, as the statistics above demonstrate—simply dismissing it is inappropriate. Finally, these same porting issues are experienced across OSs as well, and a solution that addresses the full FS portability problem would be attractive.

2.2 Existing approaches and shortcomings

User-level file systems. 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, 21, 36]. This is done either through a small kernel module that reflects FS calls into user-space [4, 36, 14] or through a loopback NFS server that leverages existing kernel NFS client support [6].

User-level FSs are not sufficient, for several reasons. First, a user-level FS solution would provide no help for existing kernel-level FS implementations. Second, user-level FSs still depend on the kernel to provide low-level services such as device access, networking, and memory management. Changes to the behavior of these components will still affect a user-level FS. For instance, §2.1's Memory Pressure example would not be solved by user-level FSs. Third, user-level FSs can deadlock since most OSs were not designed to robustly support a user-level FS under low-memory situations [21]. Such deadlocks can be avoided by using a purely event-driven structure, as the SFS toolkit [21] does, but at the cost of restricting implementer flexibility. Fourth, when using interfaces not explicitly designed for user-level FSs, such as NFS loopback, user-level FS semantics are limited by the information (e.g., no `close` calls) and control (e.g., NFS's weak cache consistency) available to them. Fifth, user-level FSs provide no assistance with user-space differences, such as shared library availability and OS configuration file formats and locations.

Rump allows the execution of unmodified NetBSD kernel file systems in user-space, by reimplementing the necessary internal OS interfaces in user-space [16]. Because this approach essentially adds a library on top of existing user-space support, it suffers from all but the first of the user-level file system deficiencies. Furthermore, although rump accommodates unmodified kernel-level file systems, it does so at a cost: reimplementing the internal OS interfaces that file systems rely on. These interfaces are much broader than the VFS interface. To achieve inter-OS operability, this reimplementation must be performed for each OS (version). Also, conflicts between kernel and user-space interfaces can pose problems. For example, rump's NFS server required modification due to conflicts between the kernel and user-space RPC portmapper and NFS mount protocol daemon.

Language-based approaches. FiST [39] provides an alternative to portable FS implementation, via a specialized language for FS developers. The FiST compiler generates OS-specific kernel modules. Given detailed information about all relevant in-kernel interfaces, updated for each OS version, FiST could address inter-version syntax changes. But, FiST was not designed to offer assistance with policy and semantic changes. Also, a specialized language is unlikely to be adopted unless it is expressive enough to address all desirable control, which is far from a solved problem.

Coccinelle [23] is a program transformation tool that automatically updates Linux device drivers after API changes. While Coccinelle could handle some of the interface syntax changes that we described, like FiST, it would be unable to mitigate the policy and semantic problems. The latter are OS design artifacts that require much more intrusive FS changes.

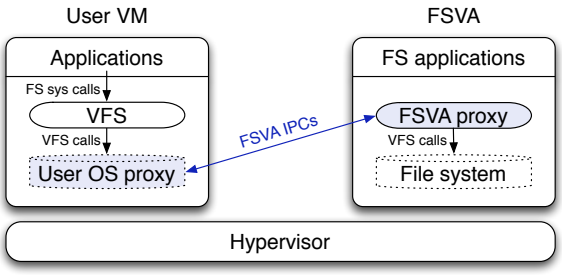

Figure 2. FSVA architecture. An FS and its (optional) management applications run in a dedicated VM. An FS-agnostic proxy running in the client OS and FSVA pass VFS calls via an efficient IPC transport.

Software engineering approaches. The software engineering community has studied the general problem of variability management. Software product lines (SPL) [9] is a technique that advocates a disciplined approach to finding and reusing common functionality (and interfaces) among related products. In a single vendor environment, or when multiple vendors agree on a common interface, SPL can be effective.

Unfortunately, different OS vendors (and even different releases of the same OS) have failed to agree on a common, comprehensive VFS interface. Different design choices, backward compatibility, and tight coupling to other (changing) OS components (e.g., the virtual memory subsystem) mean that the differences in OSs' VFS interfaces and syntax are here to stay. Overcoming policy and semantic differences is even more challenging. Some differences (e.g., §2.1's Stack Size example) arise from entirely non-FS-related OS design choices.

2.3 FSVAs = VM-level FSs

The FSVA approach promoted here is similar in spirit to user-level FSs. As before, a small FS-agnostic 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 leverages virtualization to address the compatibility challenges discussed above. In contrast to user-level FSs, FSVAs support legacy FS implementations and permit an FS to use OS-specific functionality (e.g., RDMA) while still supporting multiple OSs. Furthermore, FSVAs fully isolate the FS from user OSs and thus overcome the policy and semantic challenges described in §2.1.

Figure 2 illustrates the FSVA architecture. User applications run in a user's preferred OS.[1] An FS implementation executes in a VM running the FS vendor's preferred OS. In the user OS, an FS-independent proxy registers as an FS with the VFS layer. The user OS proxy sends all VFS calls to a proxy in the FSVA that sends the VFS calls to the actual FS implementation. The two proxies translate to/from a common VFS interface and cooperate to maintain OS and VM features such as a unified buffer cache (§3.2) and migration (§3.3).

Using an FSVA, an FS developer can tune and debug her implementation to a single OS version without concern for the user's particular OS (version). The FS will be insulated from both in-kernel and user-space differences in user OSs, because it interacts with just the one FSVA OS version. Even issues like the poor handling of memory pressure and write-back can be addressed by simply not using such a kernel in the FSVA—the FS implementer can choose an OS to suit the FS, rather than being forced to work with a user's chosen OS.

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

2.4 Viability

For the FSVA approach to succeed, the FSVA interface must be stable. Otherwise, FSVAs would merely shift the location of the changing-interfaces problem. Towards that end, we designed a minimal FSVA interface. Since the majority of interface and policy changes occur in internal OS functionality (§2.1), not at the core VFS interface, we expect FSVA interface stability to be possible. NFS provides a successful model of a constant FS interface that has enjoyed wide OS support — though, as discussed in §2.2, it is inadequate for our purposes.

The user OS and FSVA proxies are dependent on the hypervisor interface. Consequently, a proliferation of hypervisors could make it difficult for OS vendors to support the proxies for every hypervisor. Fortunately, there are only a few widely-used hypervisors. Furthermore, the hypervisor-specific code is a quarter of the user OS and FSVA proxies (about 2200 SLOC, as measured by SLOCCount). Given the (necessarily) thin hypervisor interface, it is unlikely that the hypervisor-interfacing RPC code will significantly change over time. Thus, we believe it is reasonable to expect OS vendors to support common hypervisors.

FSVAs do not preclude an FS developer from porting the FS to a different OS (version). Indeed, he/she might still do so to get new features, for improved performance, or for OS bug fixes. But, FSVAs enable such porting to occur at the FS developer's pace, not at the users' pace. The FS developer can skip porting to most OSs and select a new stable OS (version) when desired.

3 Design

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

No FS changes for correctness To simplify adoption and deployment, FS developers should not have to modify their FS to run in an FSVA. But, although changes are not required for correctness (i.e., to maintain FS semantics), we allow optional FS changes in order to gain performance.

Generality The FSVA interface should be OS- and FS-agnostic. It should not make assumptions about OS internals or FS behavior.

Maintain OS and VM features Support existing user OS features such as a unified buffer cache, and memory mapping. Applications should not be aware of the FSVA separation. Existing virtualization features such as migration, checkpointing, and performance isolation should not be adversely affected.

Minimal OS and VMM changes To encourage OS vendor adoption, the user OS and FSVA proxies should require few changes to the OS. Similarly, any VMM changes should be minimal.

Efficiency FSVAs should impose minimal overheads.

Together, the above goals allows FS developers to use FSVAs, knowing that the OS-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.) These goals also serve to encourage adoption by users and OS and VMM vendors.

Two major design decisions follow from our goals. First, to maintain FS semantics for unmodified FSs, all VFS calls are passed from the user OS to the FSVA; by default, no user OS caching is performed (§3.1). Second, to maintain virtualization features in the presence of multiple user VMs, each user VM is given its own FSVA; FSVAs are not shared among user VMs (§3.3).

3.1 FSVA interface

VFS-like interface. Our goals dictate a VFS-like interface between the proxies: this is the most direct interface to existing FSs. 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 calls to support a unified buffer cache and migration. Table 1 lists the FSVA interface.

What has been left out of the FSVA interface is notable: virtual memory interactions, data and metadata caching, device access, memory allocation, locking, preemption policy, and threading. It is precisely these aspects that change most across OSs (versions) and cause the most grief for FS developers. The spartan FSVA interface ensures that it can remain constant among OSs and across OS revisions. The limited FSVA interface does not constrain the functionality of the user OSs or the FS. OS developers are free to change internal OS interfaces and implementation, as long as they maintain the proxies.

Passing all VFS calls. Our goal of a generic architecture that maintains FS semantics for unmodified FSs precludes any FS-independent user OS caching. Although avoiding calls into the FSVA (e.g., read hits and write-backs) would improve performance, embedding such functionality in the user OS proxy decreases generality and couples 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 for consistency or reliability. Similar problems arise for read caching in the user OS proxy: callback schemes would be needed for consistency, unless shared memory were used; but a shared memory metadata cache would force the two OSs to use a common format for their cached metadata, requiring intrusive OS changes.

Thus, for correctness (i.e., maintaining FS semantics), all VFS calls are sent to the FSVA by default. But, FSVA-optimized FSs can choose to override this behavior for performance reasons. For example, §4.4 discusses how letting the user OS handle the `permission` access control VFS call can make a significant performance improvement without affecting FS semantics.

3.2 Maintaining OS features

Metadata duplication. Many OS components expect in-memory file metadata such as inodes or directory entries (e.g., for open files or executing programs). Therefore, the user OS proxy creates those data structures in the user OS. The FSVA will also contain metadata, to support the FS and FSVA OS operations. Thus, metadata exists in both OSs. Note that the user OS metadata is minimal: the user OS proxy creates basic inodes and directory entries, but any FS-specific "extra" metadata (e.g., block allocation maps) is stored only in the FSVA. This follows from the FS-agnostic FSVA design — the proxies are not aware of FS-specific metadata.

Metadata duplication can be avoided through invasive OS changes to wrap metadata access. But, practically, this would complicate the adoption of the user OS proxy by OS vendors. Given that inodes and directory entries are small data structures, we opted for duplication. As we describe below, data pages are not duplicated.

For distributed FSs with cache consistency callbacks, a user OS might contain stale metadata. For example, an open file's attributes may be updated in the FSVA through a cache consistency callback. But, this inconsistency will not be visible to the user application. OSs already call into the FS in response to application operations that require up-to-date metadata. This will cause the user OS proxy to retrieve the updated metadata from the FSVA.

Unified buffer cache. Early Unix OSs had separate caches for virtual memory pages and file

system data. This had data and control disadvantages. First, disk blocks were sometimes duplicated in both caches. Second, the lack of a single eviction policy led to suboptimal cache partitioning. Unified buffer caches (UBCs) fix both problems [15, 31]. A single cache stores both virtual memory pages and FS data, avoiding copies and enabling a single eviction policy.

An analogous problem exists for FSVAs: separate caching between the user OS and FSVA OS. Without extensive OS changes, we cannot coalesce the two OSs' caches into a single cache — the OSs 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 (to obtain a single eviction policy). The user OS and FSVA proxies maintain this illusion transparently to the two OSs.

Providing a single eviction policy is complicated since 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 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 frequency.

The semantic gap between the two caches can be bridged by informing one of the OSs of the other OS's memory allocations and accesses. To cleanly support multiple FSVAs and to preserve the user OS's cache eviction semantics and performance, we chose to inform the user OS of the FSVA's memory allocations and accesses. Thus, the user OS controls the eviction policy.

The FSVA proxy registers callbacks with the FSVA buffer cache's allocation and access routines. When the FSVA proxy observes that a new page is inserted into the buffer cache, it makes a hypercall to grow the FSVA by a single page. On every response to the user OS, the FSVA proxy piggybacks page allocation and access information. On receiving a page allocation message, the user OS proxy returns a page to the hypervisor, thereby balancing the memory usage among the OSs. Furthermore, the user OS proxy allocates a *ghost page* [11, 24] in its UBC. Conceptually, the ghost page is an entry in the UBC's LRU lists that lacks a physical backing page.

On receiving a page access message, the user OS proxy calls the OS's "page accessed" function to update the ghost page's location in the OS's LRU lists. Thus, the ghost page serves as a placeholder in the user OS's UBC for the FSVA's buffer cache page. When the user OS later decides to evict this ghost page, the user OS proxy grows by a page, informs the FSVA that it should decrease its buffer cache by a corresponding page, and the FSVA returns a page to the hypervisor. The net result is a coupling of the two OSs' UBCs and a single inter-VM cache eviction policy.

The inter-VM UBC serves to optimally size the two VMs' memory sizes, based on the virtual memory workload in the user OS and the buffer cache workload in the FSVA. Note that VM ballooning [34] and page deduplication [34] are orthogonal to inter-VM UBC — these mechanisms contain heuristics for deciding on an optimal VM size, while our inter-VM UBC algorithm uses the user OS's specific UBC cache eviction policy.

Our design choice of a single FSVA per user VM (§3.3) greatly simplifies 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.

3.3 Maintaining VM features

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. Indeed, POFS and XenFS use this single FS server approach [25, 20].

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 (§3.2), performance isolation, and user VM migration (§3.3). In describing those topics, we discuss how a shared FSVA would complicate these features.

Migration. One 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 [13], 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 provide idempotent operations.

FSVAs complicate migration. In contrast to driver VMs, FSVAs can contain large state on behalf of a user VM and the FSVA interface is non-idempotent. To allow unmodified FSs running in an FSVA to support 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. Another complication is that shared memory pages (e.g., for the request/response ring and memory-mapped I/O) will likely have different physical page mappings after migration. To address these two issues, the user OS and FSVA proxies transparently restore the shared memory mappings and retransmit any pending requests and responses that were lost during the IPC layer teardown. Moreover, we retain live migration's low downtime by synchronizing the two VMs' background transfer and suspend/resume. More details are available in [1].

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

3.4 Miscellaneous

Virtualization requirements. The above design requires two basic capabilities from a hypervisor: inter-VM shared memory and event notification. Popular hypervisors provide these [3, 32]. The use of paravirtualization [3], software virtualization [33], or hardware-assisted virtualization does not affect the above design.

Security. Maintaining the user OS's security checks and policies is required in order to maintain the same applications semantics. Unix OSs perform access control in the VFS layer. Since the user OS 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 discussed in §3.5.

Locking. Similar to security checking, Unix OSs usually perform file and directory locking in the generic VFS layer. Consequently, the user OS 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.

3.5 Limitations

Out-of-VFS-band state. The FSVA design fails to capture out-of-VFS-band FS state. For example, NFSv4 uses Kerberos authentication. With Kerberos, a user runs a program to obtain credentials, which are stored in `/tmp` on a per-process-group basis. The NFSv4 VFS handlers retrieve those Kerberos credentials. To preserve the applications' authentication semantics, the use of Kerberos authentication in NFSv4 requires the credentials to be copied from the user OS to the FSVA. Since Kerberos is also used by other FSs, the user OS 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 file naming, permission semantics, and directory notifications. So, we envision a single FSVA interface for every "OS type." This paper focuses on an FSVA interface for Unix OSs, which tend to share similar VFS interfaces [18] and POSIX semantics. 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 memory overhead for an FSVA, due to an extra OS and metadata duplication. Since the FS vendor is likely to use only a small subset of the OS, and they distribute a single FSVA, it is feasible for them to fine-tune the OS leading to a small OS image. Nevertheless, the FSVA architecture may not be appropriate for environments with severe memory pressure. §5.5 quantifies this memory overhead.

4 Implementation

We used the Xen hypervisor [3] for our FSVA prototype. To demonstrate FS portability, we implemented the user OS and FSVA proxies for two different Linux kernels: 2.6.18 (released in September 2006) and 2.6.28 (released in December 2008). We also implemented the FSVA proxy for NetBSD 5.99.5 — but that port currently lacks UBC and migration support. We are also currently working on a VMware port.

An FSVA runs as an unprivileged VM. The IPC layer closely resembles Xen's block and network drivers' IPC layers, consisting of a shared memory region (containing an asynchronous I/O ring of requests and responses) and an event notification channel (for inter-VM signaling).

Most of our code is implemented in user OS and FSVA kernel modules. But, we had to make a number of 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 OS 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. We also modified the writeback code as described in §4.2. Third, to support migration, we modified the hypervisor to zero out page table entries that point to another VM at migration time. In total, these three changes constituted less than 100 SLOC.

The Linux user OS and FSVA proxies contain ∼5300 and ∼3500 SLOC, respectively, as measured by SLOCCount. Of the sum, ∼2200 SLOC belong to the migration-supporting IPC layer, and ∼700 SLOC belong to the UBC code. As a reference point, the Linux NFSv3 client code is ∼13,000 SLOC. The NetBSD FSVA proxy contains ∼2500 SLOC — recall it currently lacks UBC and migration support.

Type	Operations
Mount	`mount, unmount`
Metadata	`getattr, setattr, create, lookup, mkdir, rmdir, link, unlink, readdir, truncate, rename, symlink, readlink, dirty_inode, write_inode`
File ops	`open, release, seek`
Data	`read, write, map_page, unmap_page`
Misc.	`dentry_validate, dentry_release, flush, fsync, permission`
UBC	`invalidate_page, evict_page`
Migration	`restore_grants`

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.

4.1 Inter-proxy interface

The majority of VFS operations have a simple implementation structure. The user OS proxy's VFS handler finds a free slot on the IPC ring, encodes the operation and its arguments 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 OS proxy does not perform any memory allocations in its IPC path.

Table 1 lists the interface between the user OS and FSVA proxies. Most of the IPCs correspond to VFS calls such as `mount`, `getattr`, and `read`. As described below, there is also an IPC to support migration and two IPCs 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 OS proxy creates a sequence of *grants* for the application buffer — each grant covers one page — using Xen's shared memory facility. No hypercalls are involved in this operation. The grants are then passed in the I/O IPC. The FSVA proxy maps the grants into the FSVA address space using Xen hypercalls, calls the FS to perform I/O directly to/from the buffer, unmaps the grants using Xen hypercalls, and sends the I/O response to the user. The user OS proxy then can recycle the grants. As an optimization, if less than 4 KB of data is read or written, data is copied back and forth using *trampoline* buffers — pages that are shared during bootstrap — as the cost of the shared memory hypercalls is not amortized over the small access size (see 5.4).

Memory mapped I/O is handled in a similar fashion, except that the roles of grant issuer and user are reversed. When an application memory access causes the OS page fault handler to read a FS page, the user OS proxy performs a `map_page` IPC to the FSVA. In response, the FSVA proxy calls the FS to bring the relevant page into the buffer cache, pins the page, and returns a grant for the page. The user OS proxy then maps that grant into its buffer cache. The grant is unmapped once the user OS evicts the page.

4.2 Unified buffer cache

To maintain a UBC, the user OS proxy must be notified of page allocations and accesses in the FSVA. We added hooks to Linux to inform the FSVA of these events. When either event occurs, the FSVA proxy queues a notification. A list of these notifications are piggybacked to the user OS 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 OS, not the FSVA, page table entries to be updated.)

When the client receives piggybacked information that a page has been added to the page cache at the FSVA, it creates a copy of the metadata associated with the page in its own page cache — in the case of Linux, it allocates a `struct page` corresponding to the page at the FSVA. Piggybacked information about page accesses at the FSVA cause the client VM to update the LRU location of each `struct page` corresponding to the pages that have been accessed.

When the client experiences memory pressure, it thus has visibility into both its own pages and the pages belonging to the FSVA, and the operating system policy will dictate which pages to evict without regard to the fact that some do not necessarily exist in the client's address space. When a page belonging to the FSVA is chosen for eviction, the client sends an evict message to the FSVA. Because the client's memory reservation is charged for each page cache page used by the FSVA so that the sum of memory being used by both machines is kept under a specified limit, evicting a page which may not even be resident at the client will still help to relieve its memory pressure.

The FSVA is not allowed to evict pages on its own initiative, to prevent interference with the client's eviction policy. This is accomplished by locking or increasing the reference count on file cache pages at the FSVA to prevent them from being selected for eviction by the FSVA's operating system under normal circumstances. The locking is done in a way that does not interfere with circumstances where the FSVA must evict pages for correctness reasons, such as invalidating pages past the end of a file that has been truncated, or invalidating pages for a network file system whose connection has been lost and for which consistency can no longer be assured. In these cases, the FSVA sends an interrupt to the client VM advising it of the need to evict and does not proceed until the client acknowledges it. This not only allows for the proper accounting at both VMs, but in the case that the client has actually mapped the page, ensures it does not access the page after it has become invalid (for instance, after truncate deallocates the page, it may be reused for something else, so client applications must not continue using it and truncate must not actually deallocate the page until the client has acknowledged releasing it).

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.

A subtle UBC side-effect is that decreasing the number of FSVA free pages affects the dirty page writeback rate. To maintain the same writeback behavior, we have modified the FSVA function that determines the writeback rate such that it uses the user OS's number of free pages; this value is piggybacked on every request.

While the majority of FSVA memory allocations occur in the buffer cache, metadata allocations (e.g., for inodes and directory entries) must increase the FSVA memory. Otherwise, the FSVA will evict buffer cache pages, decreasing performance. We continuously monitor the size of the Linux "slab" — where metadata is allocated — and grow (shrink) the FSVA as the slab grows (shrinks). The change in slab size is piggybacked on responses and the user OS changes its size accordingly.

4.3 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 IPC connection and the shared memory mappings must be reestablished. Third, in-flight requests and responses that were affected by the move must be reexecuted.

We modified Xen's migration facility to simultaneously copy two VMs' memory images. To maintain live migration's low downtime, we synchronize the background transfer of the two images and the suspend/resume events. Since the user VM depends 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 OS and FSVA proxies must reestablish their IPC connection and shared memory mappings. 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 OS proxy performs a special `restore_grants` IPC to retrieve this list from the FSVA.

When a user VM is resumed, its applications may attempt to access a 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 migration code to zero out user VM page table entries that point to another VM. So, application attempts to access the page will cause an ordinary page fault into Linux, and the user OS proxy will block the application until the page's mapping is reestablished.

Because the user-FSVA IPC 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 IPC 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 new request in the request ring slot corresponding to that response's original request.

4.4 Reducing communication overhead

Our design goal of supporting unmodified FSs does not come for free. It forces all VFS calls to be sent to the FSVA. In turn, FSVA performance is highly dependent on the IPC layer's performance.

There are two ways to reduce the communication overhead: decreasing the IPC cost or decreasing the IPC frequency. This section explores both alternatives.

Decreasing IPC cost. There are two components to IPC: data transfer and control transfer. Data transfer is fast (less than 1 μs) since requests and responses are small[2] and are stored in a shared memory region. Control transfer has two elements: VM-level scheduling and context switching, and signaling. If the user VM and FSVA are concurrently executing on different cores, then there are no VM-level scheduler and context switch latencies. But the two VMs must still signal each other of the pending request or response.

The standard Xen mechanism for inter-VM signaling employs *event channels* [3]. The Xen "send event" hypercall sends an inter-processor interrupt (IPI) to the CPU executing the other VM. Upon receipt of an IPI, the CPU invokes the OS's interrupt handler. This is effectively a thread context switch, since the current processor state must be saved before executing the interrupt handler thread. In Linux, the interrupt handler typically masks off other interrupts and cannot

[2]Requests and responses are 512 bytes, including piggybacked UBC messages. Data operations (e.g., `read` and `readdir`) use additional shared memory.

sleep. Thus, the interrupt handler is not capable of executing general-purpose kernel code that may block. The Xen event channel interrupt handler signals a worker thread, which then handles the operation. This involves a second thread context switch. In a Linux 64-bit x86 environment, a thread context switch costs $\sim$3.5 μs. Thus, a *one-way* inter-VM signal costs 7 μs in thread switch times. There are also additional overheads in sending the IPI ($\sim$2 μs).

The Xen event channel mechanism was designed for I/O devices, in which a two-way IPC signaling overhead of 18 μs would be insignificant when compared to device access time. But this overhead is too high for FSVAs, where many frequent VFS operations (e.g., `getattr`, `permission`) execute in less than 1 μs.

When multiple processors are available, a well-known technique for reducing IPC cost is to use polling as a signaling mechanism. Using polling, our IPC can avoid the expensive thread context switches. This decreases the null IPC latency from 21 μs to 4 μs (§5.4). Unfortunately, polling is energy inefficient during idle periods.

Fortunately, x86 processors include instructions that provide polling-like latency with events-like energy-efficiency. These instructions were introduced to enable energy- and performance-efficient inter-process synchronization. The `monitor` and `mwait` instructions put a processor in low-energy mode until a write occurs to a specific memory address. These are privileged instructions, so we added a new Xen hypercall that wraps these instructions. The mwait-based IPC has similar latency to the polling IPC, with a slight increase due to the hypercall cost (§5.4).

Decreasing IPC frequency. Dedicating a processor core provides efficient performance (§5.3). But, dedicating a core to the FSVA may not always be feasible or desirable. We now how efficient performance can be achieved without dedicating a processor core to the FSVA.

During FSVA performance tuning, we found the `permission` VFS handler to be one of the biggest sources of performance overhead. This VFS handler is responsible for performing access control checks and is called in most FS system calls, often more than once (e.g., for every namespace component during pathname-to-inode translation). In most FSs, the FS-specific `permission` handler calls the generic OS access control function which compares the Unix user ID with the file owner ID and mode permission. Given the simplicity of this check, the IPC overhead dwarfs the VFS operation execution time. Fortunately, the Unix access control semantics is common across most Unix OSs and FSs. Thus, for many FSs, we can avoid the `permission` IPCs because the generic user OS `permission` handler is sufficient.

With the `permission` IPC eliminated, some of our benchmarks see a 30% performance improvement (§5.3), similar to the benefits of dedicating a core. Thus, relaxing our design principle of passing all VFS calls can alleviate the need to dedicate a processor core to the FSVA. Although this requires FS modifications (e.g., to indicate that a user OS VFS handler is sufficient), the performance improvements make this worthwhile.

Discussion. We believe it is reasonable to expect FS developers to make very minor changes to optimize their FS for FSVAs. For example, handling the `permission` VFS call in the user OS requires a one-line change for each FS: to tell the FSVA proxy that the user OS can use its own `permission` handler. Given the significant performance benefits for such a minor change, we recommend that FSs take such a route.

For legacy FSs or where such FS modifications are not feasible, efficient performance can still be provided by dedicating a processor core and using our optimized IPC layer.

5 Evaluation

This section evaluates our FSVA prototype. First, it describes examples of using FSVAs to address FS portability. Second, it quantifies the performance and memory overheads of our FSVA prototype. Third, it illustrates the efficacy of the inter-VM unified buffer cache and live migration support.

5.1 Experimental setup

Experiments are performed on a dual quad-core 1.86 GHz Xeon E5320 machine with 8 GB of memory, a 10K rpm 146 GB Seagate Cheetah ST3146755SS disk connected to a Fusion MPT SAS adaptor, and a 1 Gb/s Broadcom NetXtreme II BCM5708 Ethernet NIC. Our NFS server is a single quad-core 1.86 GHz Xeon E5320 machine with 4 GB of memory, a 10K rpm 73 GB Seagate Cheetah ST373455SS disk using the same Fusion SAS adaptor and Broadcom NIC, running the Linux in-kernel NFSv3 server implementation.

We used Xen version 3.4-unstable. Linux VMs run the 64-bit Debian testing distribution, with either our modified 2.6.18 kernel (based on the Xen-maintained Linux kernel tree) or 2.6.28 kernel (based on the vanilla Linux repository). We compiled the Linux kernels with gcc 4.3.3, without debugging symbols or checks. NetBSD VMs run 64-bit NetBSD 5.99.5, compiled with gcc 4.1.3 with debugging symbols enabled.

By default, ext2 and ext3 randomly allocate block groups for top-level directories. This caused significant variance in our results across runs. In order to have repeatable results, we used the `oldalloc` mount option for ext2 and ext3, which forces a deterministic, but slower, block group allocation algorithm. When running benchmarks on a local FS, the FS was given a 108 GB raw disk partition. The NFS server exported an 18 GB ext2 partition (mounted with the `oldalloc` option).

Unless otherwise noted, a VM was given 2 GB of memory. When running FSVA experiments, the inter-VM unified buffer cache allowed us to specify a total of 2 GB for both the user VM and the FSVA; the user-FSVA VM pair do not benefit from any extra caching. Similarly, except when otherwise noted, the FSVA and user OS VMs were pinned to the same CPU in order to ensure comparable CPU access to the non-FSVA experiments.

5.2 Portable FSs via FSVAs

The efficacy of the FSVA architecture in addressing the FS portability problem is demonstrated with two case studies: one inter-OS and one intra-OS.

Linux user using NetBSD LFS. Linux does not include an LFS implementation. There are stale third-party in-kernel and user-level implementations, but they are not full-featured and have not been ported to modern kernel versions (!). NetBSD includes an in-kernel LFS implementation. Using an FSVA, a Linux 2.6.28 user OS can use the unmodified NetBSD LFS implementation (see Figure 1). We ran a random I/O benchmark in the Linux user OS, using a 200 MB test file and a 512 byte write unit size. When running over the NetBSD LFS FSVA, the benchmark achieved 19.4 MB/s. In contrast, when running over a Linux ext3 FSVA, the benchmark only achieved 0.44 MB/s. Such improved random write performance is the hallmark of the LFS approach.

Linux 2.6.18 user using Linux 2.6.28 ext4fs. The Linux 2.6.28 kernel (released in December 2008) includes a new FS: ext4. In contrast to its widely-used ext3 predecessor, ext4 adds extents, delayed allocation, and journal checksumming. Using FSVAs, a user OS running a Linux 2.6.18 kernel (released in September 2006) can use a Linux 2.6.28 ext4 FSVA. Compared to ext3, the ext4 FSVA provided over a 4 X improvement in Postmark performance. Thus, FSVAs enable

a Linux OS with a 2 year old kernel to gain the benefits of ext4 immediately, without having to upgrade.

5.3 Macrobenchmarks

To quantify FSVA overheads, we use three FS-intensive macrobenchmarks: Postmark, IOzone, and a Linux kernel compilation. To focus on FSVA overheads, both the user OS and the FSVA used an identical OS: Linux with a 2.6.28 kernel. Otherwise, differences in internal OS policies add variables to the comparisons. For example, eviction and write-back policies are different in the 2.6.18 and 2.6.28 kernels, and NetBSD performs fewer permission VFS calls than Linux due to its whole-pathname name cache, in contrast to Linux's per-pathname-component name cache.

We ran the macrobenchmarks over four FSs: ext2, ext3, NFS, and ReiserFS. Five system configurations were used. "baremetal" denotes a Linux OS running directly on the hardware without a hypervisor and "domU" denotes a Linux OS running as a paravirtualized Xen guest. In both cases, the FS executes "natively" in the OS kernel. "FSVA" denotes the FS running in an FSVA and the user OS sending all VFS operations to the FSVA. "FSVA-user-OS-permission" denotes the user OS directly handling the `permission` VFS call (§4.4). "FSVA-mwait" denotes the user OS and FSVA executing on separate cores with our new mwait-based IPC mechanism used for inter-VM signaling (§4.4).

Of the five system configurations, we are most interested in the performance difference between "FSVA-user-OS-permission" and "domU". This difference represents the FSVA architecture overhead when the FS is FSVA-optimized through a very small FS change. Similarly, comparing the "FSVA-mwait" and "domU" results shows the overhead of the FSVA architecture when no FS changes are made and the FSVA runs on a dedicated processor. We are also interested in the performance difference between "baremetal" and "domU"; this is the performance overhead of virtualization. We expect processor, VMM, and OS improvements to decrease this overhead over time, as virtualization continues its increasing adoption.

Each experiment was run three times; means and standard deviations are shown. Before each experiment, the FS partition was reformatted and caches were flushed.

Postmark. The Postmark benchmark measures performance for small file workloads akin to e-mail and netnews [17]. It measures the number of transactions per second, where a transaction is either a file create or file delete, paired with either a read or an append. Files are created with sizes randomly varying from 500 bytes to 9.77 KB. Appends use access sizes that randomly vary from 1 byte to the file size. Reads access the entire file. Default parameters were used, except for benchmark sizing: 50,000 files, 50,000 transactions, and 224 subdirectories.

Figure 3 shows that FSVAs result in less than 10% reduction in Postmark performance for all tested FSs, compared to the native in-kernel domU results, when the `permission` VFS called is handled in the user OS or when a processor core is dedicated to the FSVA. ext2 and ext3 have much faster absolute performance, and thus the absence of either optimization leads to greater relative overhead (20% and 23%, respectively) than the other file systems, because the IPC latency has a constant cost. The FSVA overhead for NFS and ReiserFS is less than 5% even without either optimization. Virtualization adds at most 10% overhead compared to the baremetal performance.

IOzone. The IOzone benchmark supports a wide range of sequential/random workloads [10]. We used IOzone to measure sequential I/O performance. A 10 GB file was sequentially written and read, using 64 KB record sizes. The file was much larger than the VM memory size, so the numbers reflect out-of-(FSVA)-cache performance.

For each FS, there was less than 2.5% difference among the various configurations. These results indicate that virtualization and use of FSVAs do not impact streaming I/O throughput,

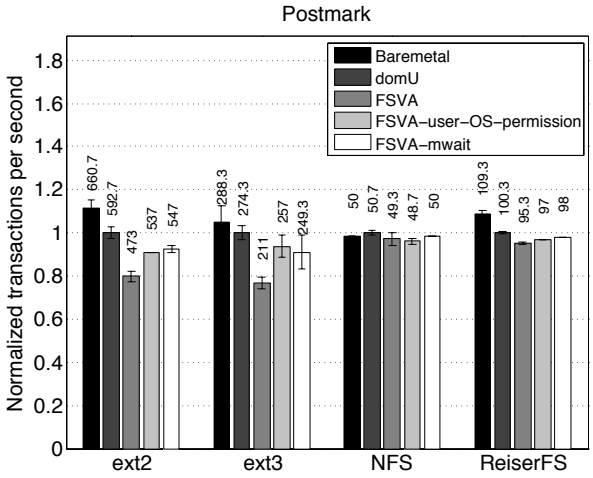

Figure 3. Postmark results, normalized to domU.

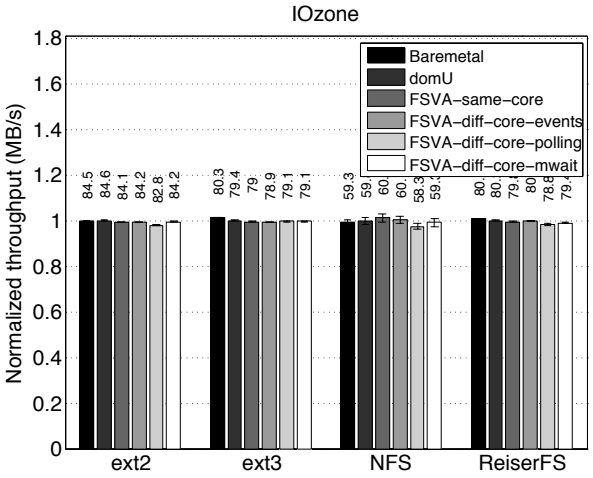

Figure 4. IOzone results, normalized to domU.

even when the user VM and FSVA share a single CPU core and the `permission` VFS call is always sent to the FSVA.

Linux kernel build. This benchmark consists of building the Linux 2.6.28 kernel. The kernel archive was copied to the FS, unarchived, and compiled. Approximately 1000 source files were compiled in our kernel configuration file. (This benchmark will be made available upon publication.)

Figure 5 shows the results. Virtualization adds substantial overhead (6–18%) to the Linux kernel compilation, due to the many hypercalls involved with the frequent program execution. When using FSVAs, the overhead varies significantly based on the configuration. With a separate CPU core and mwait, the overhead is ≤7%. For the single core configuration where all VFS calls are sent to the FSVA, 20%–40% slowdowns occur. The culprit for these slowdowns is the frequent `permission` IPCs. For example, for the ext3 case, `permission` IPCs account for 60% of the 9,508,636 IPCs. For all FSs tested, the `permission` VFS handler is very simple: it calls the generic OS access control handler. Thus, IPC overheads are highlighted. By handling the `permission` VFS call in the user OS, the FSVA overhead for the single-core configuration is less than 15% for all FSs.

5.4 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 median of ten runs. The send event operation refers to sending an event notification to another VM. A VM mapping another VM's grant performs the map grant hypercall, and then performs an unmap grant hypercall once it is done with the page. 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.

A traditional Xen IPC requires two event notifications, each consisting of an inter-processor interrupt (IPI) and two thread switches (§4.4). Those four operations correspond to 18 μs of the 21.21 μs null IPC latency we observed. The remainder of the IPC latency goes towards locking the shared IPC ring, copying the request and response data structures onto the ring, and other

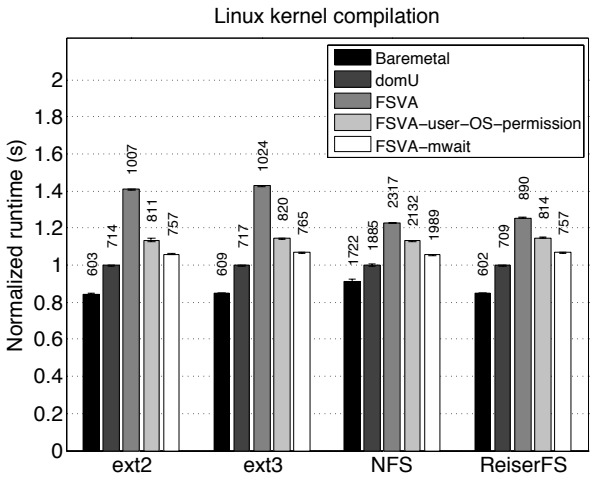

Figure 5. Linux kernel build runtime, normalized to domU.

Operation	**Latency (μs)**
Null hypercall	0.24
Send event (hypercall+IPI)	2.09
Create grant	0.21
Destroy grant	0.36
Map grant	1.99
Unmap grant	2.19
4 KB memcpy	0.80
Thread switch	3.52
Null IPC (diff core)	21.21
Null IPC (same core)	16.70
Null IPC (diff core, polling)	4.04
Null IPC (diff core, mwait)	4.34

Table 2. FSVA microbenchmarks. Latencies are in μs.

miscellaneous operations. When two VMs are pinned to the same core, Xen avoids sending an IPI and only does a VM context switch, leading to a slightly faster IPC (16.70 μs. The OS thread switches still occur, since the OS still executes its normal interrupt-handling routine once the VM is scheduled.

When inter-VM signaling is achieved by polling or our new mwait hypercall, the null IPC latency drops to 4.04 μs and 4.34 μs, respectively. The extra latency for the mwait-based IPC is due to the cost of a hypercall. Thus, avoiding the VM and thread context switches is crucial in reducing the IPC latency, and our mwait-based IPC has similar performance to the polling-based IPC but without its energy inefficiency.

5.5 Memory overhead

There is a memory overhead to using FSVAs, with two components: memory for the FSVA OS image and memory for duplicated metadata. Of course, the particular values for this memory overhead will vary depending on the particular OS image and the amount of metadata in use. As concrete examples, we report the memory overhead when running the reported macrobenchmarks.

The Linux 2.6.28 FSVA uses 72 MB of memory for the OS image. Our FSVA proxy sets aside 64 MB of memory for an initial extra reservation. Then, during benchmark execution, we observed 112–136 MB of additional memory allocated for metadata. Thus, the total memory overhead was 248–272 MB. This can be reduced in two ways. First, the Linux kernel can be fine-tuned and extra functionality can be removed. For benchmarking purposes, we used the same Linux 2.6.28 kernel in all experiments. But, when running as a Xen paravirtualized guest, the kernel can be substantially trimmed down. Second, as described in the §4.2, we currently do not put pressure on the size of the metadata allocated in the FSVA.

5.6 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. Both VMs are started with 1 GB of memory. Once the user and FSVA kernel modules are loaded,

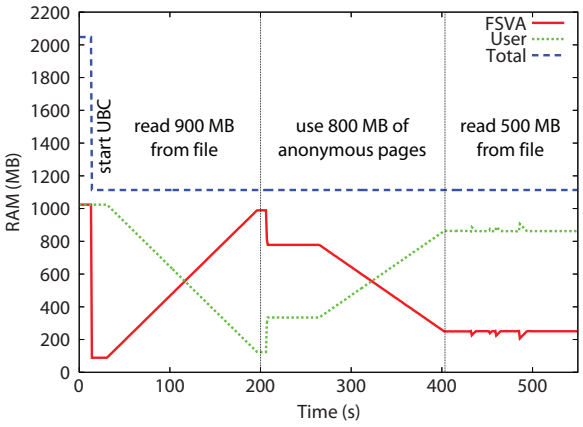

Figure 6. Unified buffer cache. This figures shows the amount of memory consumed by the user and FSVA VMs. As applications shift their memory access pattern between file system and virtual memory usage, the unified buffer cache dynamically allocates memory among the two VMs while maintaining a constant total memory allocation.

however, the FSVA returns most of its memory to Xen, thereby limiting the overall memory usage to slightly over 1 GB.

Figure 6 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 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 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. Page eviction batching is responsible for the dips in the figure.

5.7 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, there was 0.29 s unavailability. 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 and the overhead of our

IPC layer and shared memory restoration. We believe this overhead is relatively independent of the overall memory size, but were unable to run larger migration experiments due to limitations in preallocated shadow page tables that Xen uses during migration.

6 Additional related work

File systems and VMs. Several research projects have explored running a FS in another VM, for a variety of reasons. POFS 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 [25]. VPFS builds a trusted storage facility out of untrusted legacy FSs [37]. XenFS provides a shared cache between VMs and shares a single copy-on-write FS image among VMs [20]. Our FSVA architecture adapts these ideas to address the portable FS implementation problem. The differing goals lead to many design differences. We maintain OS and virtualization features such as a unified buffer cache and migration. By default, 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 POFS and XenFS focus on using a single FS per physical machine to increase efficiency.

Parallax runs storage VMs in a shared infrastructure to provide a block-level VM storage interface that includes features such as efficient snapshotting [22]. Ventana [26] is a distributed FS that provides an FS-level VM storage interface. In contrast, FSVAs provides a FS-level interface to *existing* FS implementations that is targeted towards a single user VM.

VNFS [40] 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. 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 in different OSs by running them in a VM [19]. Soft devices simplify device-level development by reusing Xen's narrow paravirtualized device interface [35]. FSVAs share both approaches' aim of leveraging existing kernel code and simplifying OS support. In addition, FSVAs deal with a richer FS interface while retaining OS and virtualization features.

Fast inter-VM communication. Fido [5] enables zero-copy inter-VM data movement through a single shared address space. FSVAs avoid data copies by using hypervisor shared memory hypercalls. Adopting Fido's single address-space approach would eliminate the need for the shared memory hypercalls. Fido's optimization is data-centric and orthogonal to our control-centric mwait-based IPC mechanism.

7 Conclusion

FSVAs offer a solution to FS portability problems. An FS can be developed, debugged, and tuned for one OS and bundled with it in a preloaded VM (the FSVA). Users can run whatever OS they like, in a separate VM, and use the FSVA like any other FS. Case studies and other experiments show that this approach works for a range of FS implementations across distinct OSs, with minimal performance overheads and no visible semantic changes for the user OS.

References

[1] Michael Abd-El-Malek, Matthew Wachs, James Cipar, Karan Sanghi, Gregory R. Ganger, Garth A. Gibson, and Michael K. Reiter. File system virtual appliances: Portable file system implementations. Technical report, Carnegie Mellon University Parallel Data Lab, May 2009.

[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. pages 44–54, New York, NY, USA, 2007.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. pages 164–177, New York, NY, USA.

[4] Brian N. Bershad and C. Brian Pinkerton. Watchdogs: Extending the unix file system. pages 267–275, 1988.

[5] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, Lakshmi N. Bairavasundaram, Kaladhar Voruganti, and Garth R. Goodson. Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances. Berkeley, CA, 2009.

[6] Brent Callaghan and Tom Lyon. The automounter. pages 43–51, 1989.

[7] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. pages 317–327, Atlanta, GA, 2000.

[8] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. pages 273–286, Berkeley, CA, 2005.

[9] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. Boston, MA, 2001.

[10] Don Capps and William Norcott. Iozone. `http://www.iozone.org`.

[11] M. Ebling, L. Mummert, and D. Steere. Overcoming the network bottleneck in mobile computing. Santa Cruz, CA, 1994.

[12] Michael Eisler, Peter Corbett, Michael Kazar, Daniel S. Nydick, and Christopher Wagner. Data ontap gx: a scalable storage cluster. pages 23–23, Berkeley, CA, 2007.

[13] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Reconstructing i/o. Technical report, University of Cambridge, Computer Laboratory, August 2004.

[14] FUSE. Fuse: filesystem in userspace. http://fuse.sourceforge.net.

[15] R. A. Gingell, J. P. Moran, and W. A. Shannon. Virtual memory architecture in sunos. *USENIX ATC*, pages 81–94, 1987.

[16] Antti Kantee. Rump File Systems: Kernel Code Reborn. Berkeley, CA, 2009.

[17] Jeffrey Katcher. Postmark: A new file system benchmark. Technical report, October 1997.

[18] S. R. Kleiman. Vnodes: an architecture for multiple file system types in sun unix. pages 238–247.

[19] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. pages 17–30.

[20] Mark Williamson. Xenfs. http://wiki.xensource.com/xenwiki/XenFS.

[21] D. Mazieres. A toolkit for user-level file systems. June 2001.

[22] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: virtual disks for virtual machines. pages 41–54, New York, NY, 2008.

[23] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. pages 247–260, New York, NY, 2008.

[24] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. pages 79–95, New York, NY, 1995.

[25] Ben Pfaff. *Improving Virtual Hardware Interfaces*. PhD thesis, 2007.

[26] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. pages 353–366, 2006.

[27] 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.

[28] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *TOCS*, 10(1):26–52, 1992.

[29] Constantine Sapuntzakis and Monica S. Lam. Virtual appliances in the collective: a road to hassle-free computing. pages 10–10, Berkeley, CA, 2003.

[30] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. page 19, Berkeley, CA, 2002.

[31] Chuck Silvers. Ubc: an efficient unified i/o and memory caching subsystem for netbsd. pages 54–54, Berkeley, CA, 2000.

[32] VMWare. Virtual machine communication interface. http://pubs.vmware.com/vmci-sdk/index.html.

[33] VMWare. Vmware esx server product overview. http://www.vmware.com/products/vi/esx/.

[34] C. Waldspurger. Memory resource management in vmware esx server. pages 181–194, 2002.

[35] Andrew Warfield, Steven Hand, Keir Fraser, and Tim Deegan. Facilitating the development of soft devices. pages 22–22, Berkeley, CA, 2005.

[36] Neil Webber. Operating system support for portable filesystem extensions. pages 219–228, 1993.

[37] Carsten Weinhold and Hermann Härtig. Vpfs: building a virtual private file system with a small trusted computing base. pages 81–93, New York, NY, 2008.

[38] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. pages 1–17, Berkeley, CA, 2008.

[39] Erez Zadok and Jason Nieh. Fist: A language for stackable file systems. pages 55–70, 2000.

[40] Xin Zhao, Atul Prakash, Brian Noble, and Kevin Borders. Improving distributed file system performance in virtual machine environments. Technical report.