# Understanding Performance Implications of Nested File Systems in a Virtualized Environment

Duy Le[1], Hai Huang[2], and Haining Wang[1]

[1]*The College of William and Mary, Williamsburg, VA 23185, USA*
[2]*IBM T. J. Watson Research Center, Hawthorne, NY 10532 USA*

## Abstract

Virtualization allows computing resources to be utilized much more efficiently than those in traditional systems, and it is a strong driving force behind commoditizing computing infrastructure for providing cloud services. Unfortunately, the multiple layers of abstraction that virtualization introduces also complicate the proper understanding, accurate measurement, and effective management of such an environment. In this paper, we focus on one particular layer: storage virtualization, which enables a host system to map a guest VM's file system to almost any storage media. A flat file in the host file system is commonly used for this purpose. However, as we will show, when one file system (guest) runs on top of another file system (host), their nested interactions can have unexpected and significant performance implications (as much as 67% degradation). From performing experiments on 42 different combinations of guest and host file systems, we give advice on how to and how not to nest file systems.

## 1 Introduction

Virtualization has significantly improved hardware utilization, thus, allowing IT services providers to offer a wide range of application, platform and infrastructure solutions through low-cost, commoditized hardware (e.g., Cloud [1, 5, 11]). However, virtualization is a double-edged sword. Along with many benefits it brings, virtualized systems are also more complex, and thus, more difficult to understand, measure, and manage. This is often caused by layers of abstraction that virtualization introduces. One particular type of abstraction, which we use often in our virtualized environment but have not yet fully understood, is the nesting of file systems in the guest and host systems.

In a typical virtualized environment, a host maps regular files as virtual block devices to virtual machines

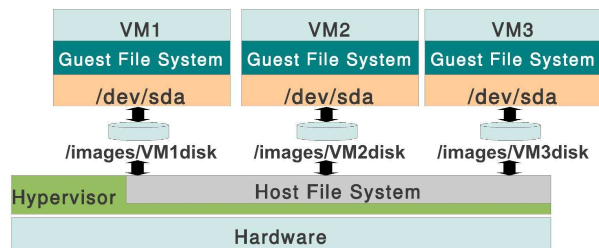


Figure 1: Scenario of nesting of file systems.

(VMs). Completely unaware of this, a VM would format the block device with a file system that it thinks is the most suitable for its particular workload. Now, we have two file systems – a host file system and a guest file system – both of which are completely unaware of the existence of the other layer. Figure 1 illustrates such a scenario. The fact that there is one file system below another complicates an already delicate situation, where file systems make certain assumptions, based on which, optimizations are made. When some of these assumptions are no longer true, these optimizations will no longer improve performance, and sometimes, will even hurt performance. For example, in the guest file system, optimizations such as placing frequently used files on outer disk cylinders for higher I/O throughput (e.g., NTFS), de-fragmenting files (e.g., QCoW [7]), and ensuring meta-data and data locality, can cause some unexpected effects when the real block allocation and placement decisions are done at a lower level (i.e., in the host).

An alternative to using files as virtual block devices is to give VMs direct access to physical disks or logical volumes. However, there are several benefits in mapping virtual block devices as files in host systems. First, using files allows storage space overcommit when they are thinly provisioned. Second, snapshotting a VM image using copy-on-write (e.g., using QCoW) is simpler at the file level than at the block level. Third, managing and maintaining VM images and snapshots as files is

also easier and more intuitive as we can leverage many existing file-based storage management tools. Moreover, the use of nested virtualization [6, 15], where VMs can act as hypervisors to create their own VMs, has recently been demonstrated to be practical in multiple types of hypervisors. As this technique encourages more layers of file systems stacking on top of one another, it would be even more important to better understand the interactions across layers and their performance implications.

In most cases, a file system is chosen over other file systems primarily based on the expected workload. However, we believe, in a virtualized environment, the guest file system should be chosen based on not only the workload but also the underlying host file system. To validate this, we conduct an extensive set of experiments using various combinations of guest and host file systems including Ext2, Ext3, Ext4, ReiserFS, XFS, and JFS. It is well understood that file systems have different performance characteristics under different workloads. Therefore, instead of comparing different file systems, we compare the same guest file system among different host file systems, and vice versa. From our experiments, we observe significant I/O performance differences. An improper combination of guest and host file systems can be disastrous to performance; but with an appropriate combination, the overhead can be negligible.

The main contributions of this paper are summarized as follows.

- A quantitative study of the interactions between guest and host file systems. We demonstrate that the virtualization abstraction at the file system level can be more detrimental to the I/O performance than it is generally believed.
- A detailed block-level analysis of different combinations of guest/host file systems. We uncover the reasons behind I/O performance variations in different file system combinations and suggest various tuning techniques to enable more efficient interactions between guest and host file systems to achieve better I/O performance.

From our experiments, we have made the following interesting observations: (1) for write-dominated workloads, journaling in the host file system could cause significant performance degradations, (2) for read-dominated workloads, nested file systems could even improve performance, and (3) nested file systems are not suitable for workloads that are sensitive to I/O latency. We believe that more work is needed to study performance implications of file systems in virtualized environments. Our work takes a first step in this direction, and we hope that these findings can help file system designers to build more adaptive file systems for virtualized environments.

The remainder of the paper is structured as follows. Section 2 surveys related works. Section 3 presents macro-benchmarks to understand the performance implications of nesting file systems under different types of workloads. Section 4 uses micro-benchmarks to dissect the interactions between guest and host file systems and their performance implications. Section 5 discusses significant consequences of nested file systems with proposed techniques to improve I/O performance. Finally, Section 6 concludes the paper.

## 2 Related Work

Virtualizing I/O, especially storage, has been proven to be much more difficult than virtualizing CPU and memory. Achieving bare-metal performance from virtualized storage devices has been the goal of many past works. One approach is to use para-virtualized I/O device drivers [26], in which, a guest OS is aware of running inside of a virtualized environment, and thus, uses a special device driver that explicitly cooperates with the hypervisor to improve I/O performance. Examples include KVM's VirtIO driver [26], Xen's para-virtualized driver [13], and VMware's guest tools [9]. Additionally, Jujjuri *et al.* [22] proposed to move the para-virtualization interface up the stack to the file system level.

The use of para-virtualized I/O device drivers is almost a de-facto standard to achieve any reasonable I/O performance, however, Yassour *et al.* [32] explored an alternative solution that gives guest direct access to physical devices to achieve near-native hardware performance. In this paper, we instead focus on the scenario where virtual disks are mapped to files rather than physical disks or volumes. As we will show, when configured correctly, the additional layers of abstraction introduce only limited overhead. On the other hand, having these abstractions can greatly ease the management of VM images.

Similar to nesting of file systems, I/O schedulers are also often used in a nested fashion, which can result in suboptimal I/O scheduling decisions. Boutcher and Chandra [17] explored different combinations of I/O schedulers in guest and host systems. They demonstrated that the worst case combination provides only 40% throughput of the best case. In our experiments, we use the best combination of I/O schedulers found in their paper but try different file system combinations, with the focus on performance variations caused only by file system artifacts. Whereas, for performance purposes, there is no benefit to performing additional I/O scheduling in the host, it has a significant impact on inter-application I/O isolation and fairness as shown in [23]. Many other works [18, 19, 25, 27] have also studied the impact of nested I/O schedulers on performance, fairness, and iso-

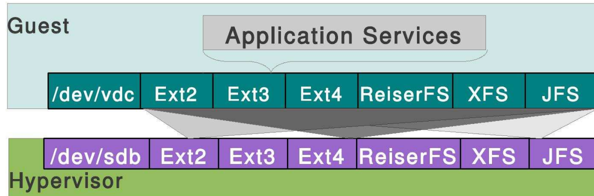


Figure 2: Setup for macro-level experimentation

lation, and these are orthogonal to our work in the file system space.

When a virtual disk is mapped to an image file, the data layout of the image file can significantly affect its performance. QCOW2 [7], VirtualBox VDI [8], and VMware VMDK [10] are some popular image formats. However, as Tang [31] pointed out, these formats unnecessarily mix the function of storage space allocation with the function of tracking dirty blocks. Tang presented an FVD image format to address this issue and demonstrated significant performance improvements for certain workloads. Various techniques [16, 20, 30] to dynamically change the data layout of image files, depending on the usage patterns, have also been proposed. Suzuki *et al.* [30] demonstrated that by co-locating data blocked used at boot time, a virtual machine can boot much faster. Bhadkamkar *et al.* [16] and Huang *et al.* [20] exploited data replication techniques to decrease the distance between temporally related data blocks to improve I/O performance. Sivathanu *et al.* [29] studied the performance effect of the image file placed at different locations of a disk.

I/O performance in storage virtualization can be impacted by many factors, such as device driver, I/O scheduler, and image format. To the best of our knowledge, this is the first work that studies the impact of the choice of file systems in guest and host systems in a virtualization environment.

## 3 Macro-benchmark Results

To better understand the performance implications caused by guest / host file system interactions, we take a systematic approach in our experimental evaluation. First, we exercise macro-benchmarks to understand the potential performance impact of nested file systems on realistic workloads, from which, we were able to observe significant performance impact. In Section 4, we use micro-benchmarks coupled with low-level I/O tracing mechanisms to investigate the underlying cause.

### 3.1 Experimental Setup

As there is no single "most common" or "best" file system to use in the hypervisor or guest VMs, we conduct our experiments using all possible combinations of popular file systems on Linux (i.e., Ext2, Ext3, Ext4, ReiserFS, XFS, and JFS) in both the hypervisor and guest VMs, as shown in Figure 2. A single x86 64-bit machine is used to run KVM [24] at the hypervisor level, and QEMU [14] is used to run guest VMs [1]. To reflect typical enterprise setting, each guest VM is allocated a single dedicated processor core. More hardware and software configuration settings are listed in Table 1.

| | **Hardware** | **Software** |
|---|---|---|
| **Host** | Pentium D 3.4GHz, 2GB RAM | Ubuntu 10.04 (2.6.32-33) |
| | 80GB WD 7200 RPM SATA (**sda**) | qemu-kvm 0.12.3 |
| | 1TB WD 7200 RPM SATA (**sdb**) | libvirt 0.9.0 |
| **Guest** | Qemu 0.9, 512MB RAM | Ubuntu 10.04 (2.6.32-33) |

Table 1: Testbed setup

The entire host OS is installed on a single disk (`sda`) while another single disk (`sdb`) is used for experiments. We create multiple equal-sized partitions from `sdb`, each corresponding to a different host file system. Each partition is then formatted using the default parameters of the host file system's `mkfs*` command and is mounted using the default parameters of `mount`. In the newly created host file system, we create a flat file and expose this flat file as the logical block device to the guest VM, which in turn, further partitions the block device, having each corresponding to a different guest file system. By default, `virtio` [26] is used as the block device driver for the guest VM and we consider `write-through` as a caching mode for all backend storages. The end result is the guest VM having access to all combinations of guest and host file systems. Table 2 shows an example of our setup: a file created on `/dev/sdb3`, which is formatted as Ext3, is exposed as a logical block device `vdc` to the guest VM, which further partitions `vdc` into `vdc2`, `vdc3`, `vdc4`, etc. for different guest file systems. Note that all disk partitions of the hypervisor (`sdb*`) and the guest (`vdc*`) are properly aligned using `fdisk` to avoid most of the block layer interference caused by misalignment problems.

In addition to the six host file systems, we also create a raw disk partition that is directly exposed to the guest VM and is labeled as *Block Device (BD)* in Table 2. This allows a guest file system to sit directly on top of a physical disk partition without the extra host file system layer. This special case is used as our baseline to demonstrate how large (or how small) of an overhead the host file system layer induces. However, there are some side effects to this particular setup, and namely, the file systems being created on outer disk cylinders will have higher I/O throughput than those created on inner cylinders. For-

[1] Similar performance variations are observed in the experiments with other hypervisors including Xen and VMWare, which are shown in Appendix.

| Host file system | | | |
|---|---|---|---|
| **Devices** | **#Blocks (x$10^6$)** | **Speed(MB/s)** | **Type** |
| sdb2 | 60.00 | 127.64 | Ext2 |
| sdb3 | 60.00 | 127.71 | Ext3 |
| sdb4 | 60.00 | 126.16 | Ext4 |
| sdb5 | 60.00 | 125.86 | ReiserFS |
| sdb6 | 60.00 | 123.47 | XFS |
| sdb7 | 60.00 | 122.23 | JFS |
| sdb8 | 60.00 | 121.35 | Block Device |

| Guest file system | | |
|---|---|---|
| **Device** | **#Blocks x$10^6$** | **Type** |
| vdc2 | 9.27 | Ext2 |
| vdc3 | 9.26 | Ext3 |
| vdc4 | 9.27 | Ext4 |
| vdc5 | 9.28 | ReiserFS |
| vdc6 | 9.27 | XFS |
| vdc7 | 9.08 | JFS |

Table 2: Physical and logical disk partitions

| Services | # Files | # Threads | File size | I/O size |
|---|---|---|---|---|
| File server | 50,000 | 50 | 128KB | 16KB-1MB |
| Web server | 50,000 | 100 | 16KB | 512KB |
| Mail server | 50,000 | 16 | 8-16KB | 16KB |
| DB server | 8 | 200 | 1GB | 2KB |

Table 3: Parameters for Filebench workloads

tunately, as each disk partition created at the hypervisor level is 60GB, only a portion of the entire disk is utilized and thus limits this effect. Table 2 also shows the results of running `hdparm` on each disk partition. The largest throughput difference between any two partitions is only about 5%, which is fairly negligible.

The choice of I/O scheduler at host and guest levels can significantly impact performance [17, 21, 27, 28]. As file system is the primary focus of this paper, we used `CFQ` scheduler in the host and `Deadline` scheduler in the guest as these schedulers were shown to be the top performers in their respective domains by Boutcher and Chandra [17].

## 3.2 Benchmarks

We use Filebench [3] to generate macro-benchmarks of different I/O transaction characteristics controlled by predefined parameters, such as the number of files to be used, average file size, and I/O buffer size. Since Filebench supports a synchronization between threads to simulate concurrent and sequential I/Os, we use this tool to create four server workloads: a file server, a web server, a mail server, and a database server. The specific parameters of each workload are listed in Table 3, showing that the experimental working set size is configured to be much larger than the size of the page cache in the VM. The detailed description of these workloads is as follows.

- **File server:** Emulates a NFS file service. File operations are a mixture of `create`, `delete`, `append`, `read`, `write`, and `attribute` on files of various sizes.

- **Web server:** Emulates a web service. File operations are dominated by reads: `open`, `read`, and `close`. Writing to the web log file is emulated by having one `append` operation per `open`.

- **Mail server:** Emulates an e-mail service. File operations are within a single directory consisting of I/O sequences such as `open/read/close`, `open/append/close`, and `delete`.

- **Database server:** Emulates the I/O characteristic of Oracle 9i. File operations are mostly `read` and `write` on small files. To simulate database logging, a stream of synchronous `writes` is used.

## 3.3 Macro-benchmark Results

Our main objective is to understand how much of a performance impact nested file systems have on different types of workloads, and whether or not the impact can be lessened or avoided. As mentioned before, we use all combinations of six popular file systems in both the hypervisor and guest VMs. For comparison purpose, we also include one additional combination, in which the hypervisor exposes a physical partition to guest VMs as a virtual block device. This results in 42 ($6 \times 7$) different combinations of storage / file system configurations.

The performance results are shown in Figures 3 and 6, in terms of I/O throughput and I/O latency, respectively. Each sub-figure consists of a left and a right side. The left side shows the performance results when the guest file systems are provisioned directly on top of raw disk partitions in the hypervisor. These are expressed in absolute numbers (i.e., MB per second for throughput or millisecond for latency) and are used as our baseline. The right side shows the relative performance (to the baseline numbers) of the guest file systems when they are provisioned as files in the host file system. In these figures, each column group represents a different storage option

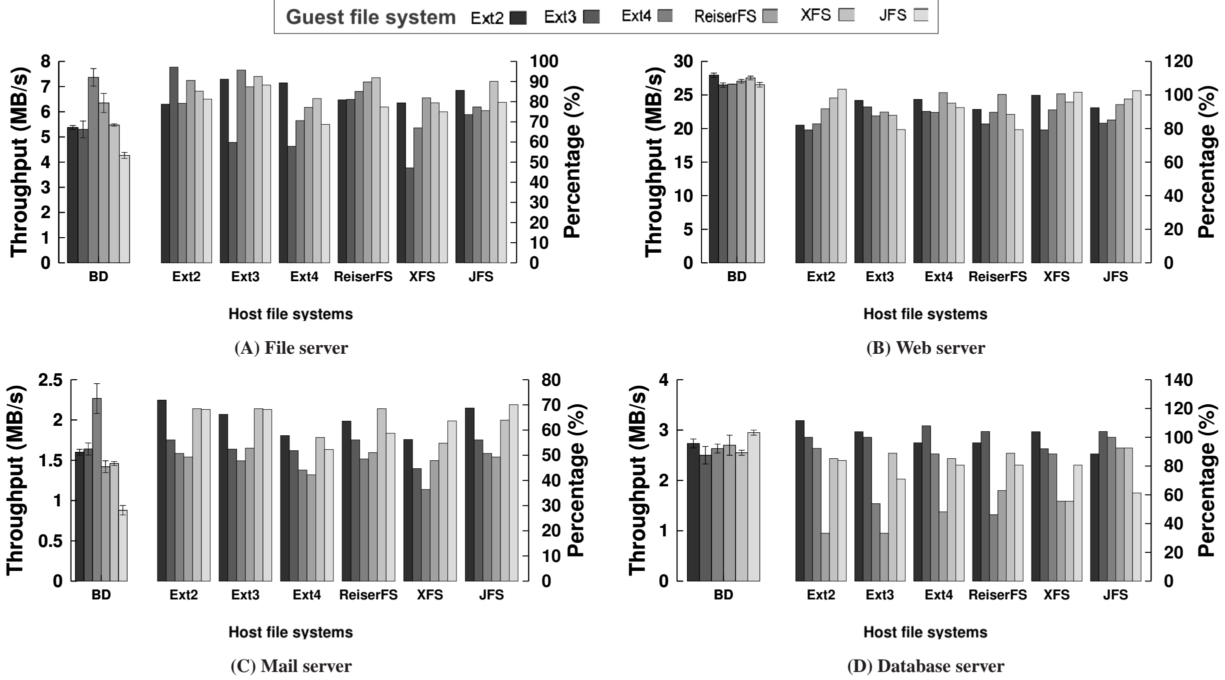


Figure 3: I/O throughput for Filebench workloads (**higher is better**)

in the hypervisor, and each column within the group represents a different storage option in the guest VM.

### 3.3.1 Throughput

The baseline numbers (leftmost column group) show the intrinsic characteristics of various file systems under different types of workloads. These characteristics indicate that some file systems are more efficient on large files than small files, while some file systems are more efficient at reading than writing. As an example, when ReiserFS runs on top of BD, its throughput under the web server workload (27.2 MB/s) is much higher than that under the mail server workload (1.4MB/s). These properties of file systems are well understood, and how one would choose which file system to use is a straightforward function of the expected I/O workload. However, in a virtualized environment where nested file systems are often used, the decision becomes more difficult. Based on the experimental results, we make the following observations:

**(1) A guest file system's performance varies significantly under different host file systems.** Figure 3(B) shows an example of the database workload. When ReiserFS runs on top of Ext2, its throughput is reduced by 67% compared to its baseline number. However, when it runs on top of JFS, its I/O performance is not impacted at all. We use coefficient of variance to quantify how differently a guest file system' performance is affected by different host file systems, which is shown in Figure 4. For

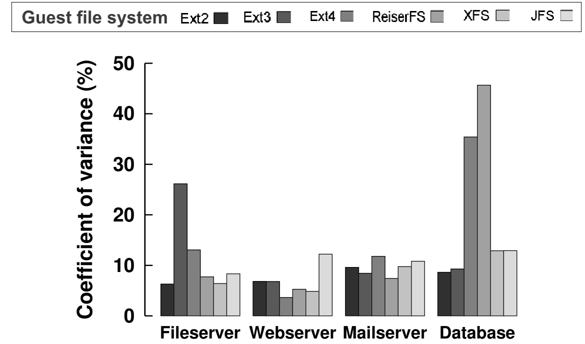


Figure 4: Coefficient of variance of guest file systems' throughput under Filebench workloads across different host file systems.

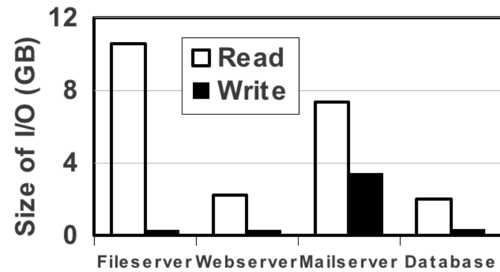


Figure 5: Total I/O transaction size of Filebench workloads

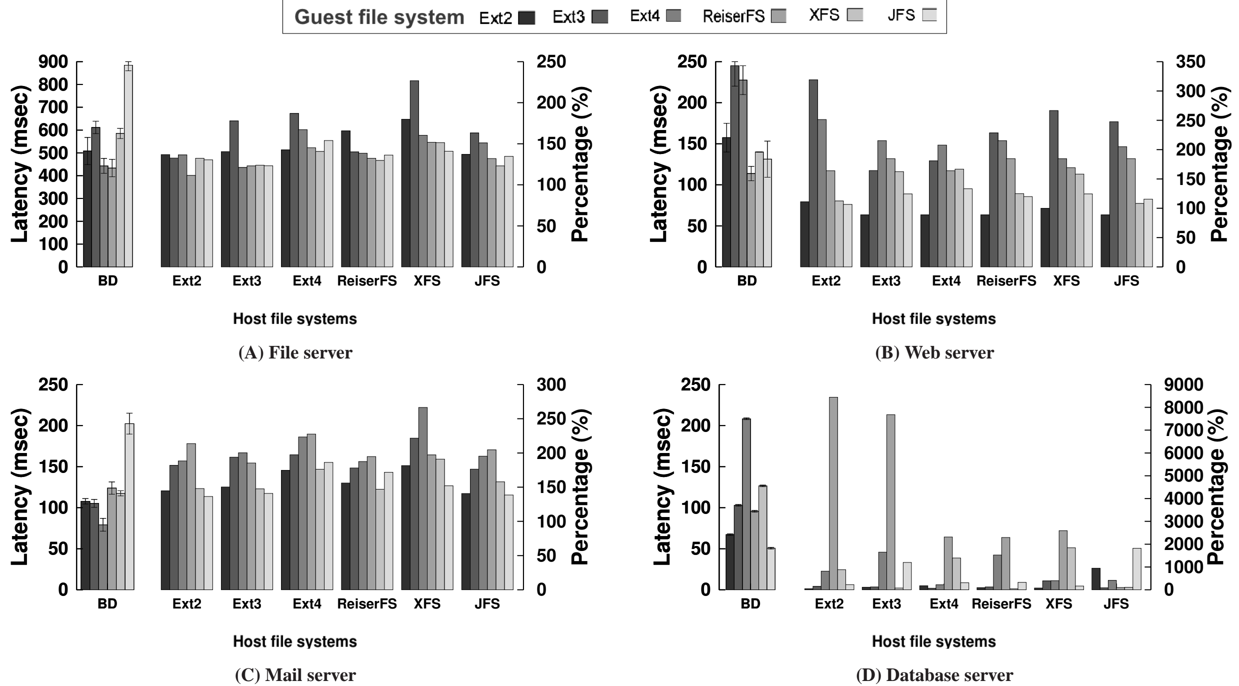


Figure 6: **I/O latency** of guest file systems under different workloads (**lower is better**)

each workload, a variance number is calculated based on relative performance values of a guest file system when it runs on top of different host file systems. Our results show that the throughput of ReiserFS experiences a large variation (45%) under the database workload, while that of Ext4 varies insignificantly (4%) under the web server workload. The large variance numbers indicate that having the right guest/host file system combination is critical to performance, and having a wrong combination can result in serious performance degradation. For instance, under the database workload, ReiserFS/Ext2 is a right combination, but ReiserFS/JFS is a wrong combination.

**(2) A host file system impacts different guest file systems' performance differently.** Similar to the previous observation, a host file system can have a different impact on different guest file systems' performance. Figure 3(A) shows an example of the file server workload. When Ext2 runs on top of Ext3, its throughput is slightly degraded by about 10%. However, when Ext3 runs on top of Ext3, the throughput is reduced by 40%. Based on results of coefficient of variance of guest file systems' throughputs shown in Figure 4, we observe that this bi-directional dependency between guest and host file systems again stresses the importance of choosing the right guest/host file system combination.

**(3) A right guest file system/host file system combination can produce minimal performance degradation.** Also based on results shown in Figure 4, one can also observe how badly performance can be impacted when a wrong combination of guest/host file system is chosen. However, it is possible to find a guest file system whose performance loss is the lowest. For example, the results of the mail server workload show that once Ext2 runs on top of Ext2, its throughput degradation is the lowest (by 46%).

**(4) The performance of nested file systems is affected much more by write than read operations.** As one can see in Figure 3, *all* the combinations of nested file systems perform poorly for the mail server workload, unlike the other three workloads. We study the detailed disk traces from these workloads by examining request queuing time, request merging, request size, etc., and find that the mail server workload is only significantly different from the others in having a much higher proportion of writes than reads, as shown in Figure 5. We will use micro-benchmarks in Section 4 to describe the reasons behind this behavior.

#### 3.3.2 Latency

The latency results are illustrated in Figure 6. Similar to I/O throughput, latency is also deteriorated when guest file systems are provisioned on top of host file systems rather than raw partitions. Whereas the impact to throughput can be minimized (for some workloads) by choosing the right combinations of guest/host file system, latency is much more sensitive to nesting of file systems. In comparison to the baseline, the latency of each guest file system varies in a certain range when it

| Description | Parameters |
|---|---|
| Total I/O size | 5 GB |
| I/O parallelism | 255 |
| Block size | 8 KB |
| I/O pattern | Random/Sequential |
| I/O mode | Native asynchronous I/O |

Table 4: FIO benchmark parameters

runs on top of different host file systems. Even for the lowest cases, latency is increased by 5-15% across the board (e.g., Ext2 guest file system under the web server workload). Coefficient of variance for latency is similar to that of throughput shown in Figure 4. However, for latency sensitive workloads, like the database workload, such a significant increase in I/O response time could be unacceptable.

## 4 Micro-benchmarks Results

We first study nested file systems using a micro-level benchmark *FIO* [4]. Based on the experimental results, we further conduct an analysis at the block layer on the guest VM and the hypervisor, respectively, using an I/O tracing mechanism [2].

### 4.1 Benchmark

We use FIO as a micro-level benchmark to examine disk I/O workloads. As a highly configurable benchmark, FIO defines a test case based on different I/O transaction characteristics, such as total I/O size, block size, number of I/O parallelism, and I/O mode. Here our focus is on the performance variation of primitive I/O operations, such as *read* and *write*. With the combination of these I/O operations and two I/O pattens, *random* and *sequential*, we design four test cases: random read, random write, sequential read, and sequential write. The specific I/O characteristics of these test cases are listed in Table 4.

### 4.2 Experimental Results

On the same testbed, the experiments are conducted with many small files, which create a 5GB of total data footprint for each workload. Figures 7 and 8 show the performance in both sequential and random I/Os. Based on the experimental results, we make two observations:

- **The performance of those workloads that are dominated by read operations is largely unaffected by nested file systems.** The performance impact is weakly dependent on guest/host file systems. More interestingly, for sequential reads, in a few scenarios, a nested file system can even improve I/O performance (e.g., by 34% for Ext3/JFS).
- **The performance of those workloads that are dominated by write operations is heavily affected by nested file systems.** The performance impact varies in both random and sequential writes, with higher variations in sequential writes. In particular, a host file system like XFS can degrade the performance by 40% for both random and sequential writes. As a result, it is important to understand the root cause of this performance impact, especially on the sequential write dominated workload.

To interpret these observations, our analysis will focus on sequential workloads and the performance implication across certain guest/host file system combinations. For this set of experiments with micro-benchmark, due to space constraints, we only concentrate on deciphering the I/O behavior of these representative file system combinations. Although only a few combinations are considered, principles used here are applicable to other combinations as well.

For sequential read workloads, we attempt to uncover the reasons behind the significant performance improvement on the *right* guest/host file system combinations. We select the combinations of **Ext3/JFS** and **Ext3/BD** for analysis. For sequential write workloads, we try to understand the root cause of the significant performance variations in the scenarios of (1) different guest file systems running on the same host file system and (2) the same guest file system operating on different host file systems. We analyze three guest file system/host file system combinations: **Ext3/ReiserFS**, **JFS/ReiserFS**, and **JFS/XFS**. Here Ext3/ReiserFS and JFS/ReiserFS are used to examine how different guest file systems can affect performance differently on the same host file system, while JFS/ReiserFS and JFS/XFS are used to examine how different host file systems can affect performance differently on the same guest file system.

### 4.3 I/O Analysis

To understand the underlying cause of the performance impact due to nesting of file systems, we use blktrace to record I/O activities at both the guest and hypervisor levels. The resulting trace files are stored on another device, thus increasing only 3-4% CPU utilization. Therefore, the interference with our benchmarks from such an I/O recoding is negligible. Blktrace keeps detailed account of each I/O request from start to finish as it goes through various I/O states (e.g., put the request onto an I/O queue, merge with an existing request, and wait on the I/O queue). The I/O states that are of interest to us in this study are described as follows.

- Q: a new I/O request is *queued* by an application.

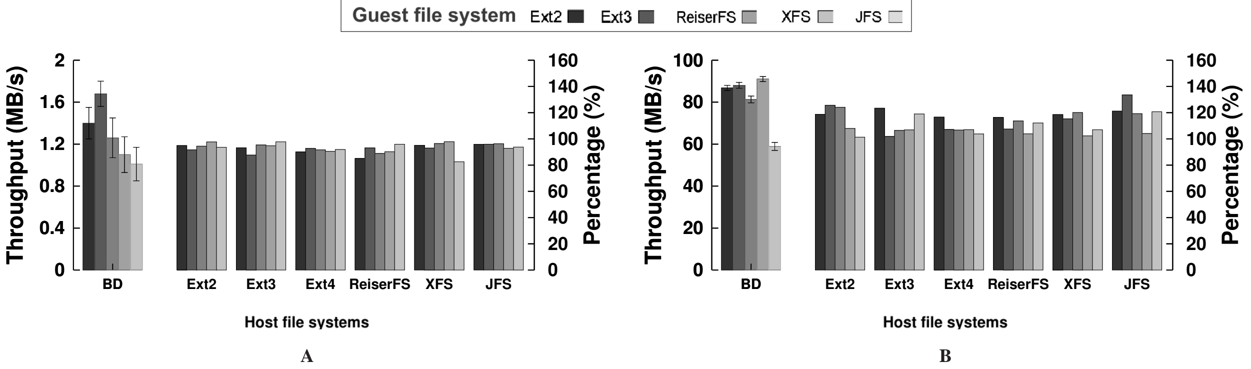


Figure 7: **I/O throughput** of guest file systems in **reading** files. (A): random and (B) sequential

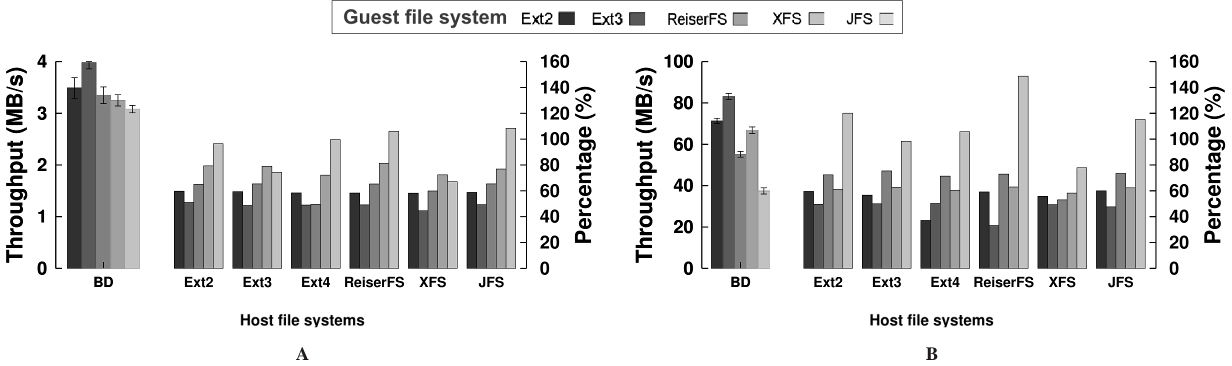


Figure 8: **I/O throughput** of guest file systems in **writing** files. (A): random and (B) sequential

- I: the I/O request is *inserted* into an I/O scheduler queue.
- D: the I/O request is being served by the *device*.
- C: the I/O request has *completed* by the device.

Blktrace records the timestamp when an I/O request enters a new state, so it is trivial to calculate the amount of time the request spends in each state (i.e., Q2I, I2D, and D2C). Here Q2I is the time it takes to insert/merge a request onto a request queue. I2D is the time it takes to idle on the request queue waiting for merging opportunities. D2C is the time it takes for the device to serve the request. The sum of Q2I, I2D, and D2C is the total processing time of an I/O request, which we denote as Q2C.

### 4.3.1 Sequential Read Workload

As mentioned in the experimental setup, the logical block device of the guest VM can be represented as either a flat file or a physical raw disk partition at the hypervisor level. However, the different representation of the guest VM's block device directly affects the number of I/O requests served at the hypervisor level. For the selected combinations of **Ext3/JFS** and **Ext3/BD**, as Figure 9 shows, the number of I/O requests served at the hypervisor's block layer is significantly lower than that at the guest's block layer. More specifically, if JFS is used as a host file system, it greatly reduces the number of queued I/O requests sent from the guest level, resulting in much fewer I/O requests served at the hypervisor level than those at the guest level. If a raw disk partition is used instead, although there is no reduction on the number of queued I/O requests, the hypervisor level's block layer also lowers the number of served I/O requests by merging queued I/O requests.

There are two root causes for these I/O behaviors: (1) the file prefetching technique at the hypervisor level, known as *readahead*, and (2) the merging activities at the hypervisor level introduced by the I/O scheduler. The detailed descriptions of these root causes are given below.

First, there are frequent accesses to both files' content and metadata in a sequential read dominated workload. To expedite this process, readahead I/O requests are issued at the kernel level of both the guest and the hypervisor. Basically, readahead I/O requests populate the page cache with data already read from the block device, so that subsequent reads from the accessed files do not block on other I/O requests. As a result, it decreases the number of accesses to the block device. In particular, at the hypervisor level, a host file system issues readahead

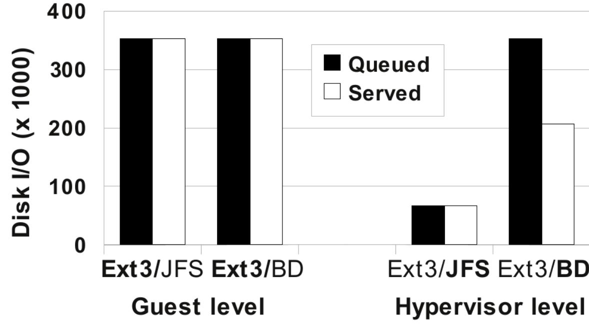


Figure 9: Disk I/Os under sequential read workload

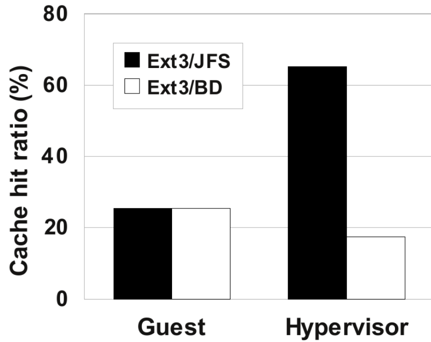


Figure 10: Cache hit ratio under sequential read workload.

requests and attempts to minimize the frequent accesses on the flat file by caching the subsequently accessed contents and metadata in the physical memory. Therefore, the I/Os served at the hypervisor level are much fewer than those at the guest level.

However, when accessing a raw disk partition, there is no readahead. Thus, for sequential workloads, a host file system outperforms a raw disk partition due to more effective caching. This discrepancy of data caching at the hypervisor level is clearly shown in Figure 10.

Second, to optimize I/O requests being served on the block device, the hypervisor's block layer attempts to reduce the number of accesses into the block device by sorting and merging queued I/O requests. However, when many I/O requests are sorted and merged, they need to stay longer in the queue than normal. For JFS (host file system), as shown in Figure 9, due to the effective caching, much fewer I/O requests are sent to the disk, and thus much fewer sorting/merging activities occur at the I/O queue. However, when a raw partition is used, much more I/O requests need to be sorted/merged. The sorting/merging activities cause a higher idle time (I2D) for I/O requests being served on the block device than those on the JFS (host file system). This behavior is depicted in Figure 11 (hypervisor level).

**Remark:** When a flat file is used as a guest VM's logical block device, sequential read dominated workloads

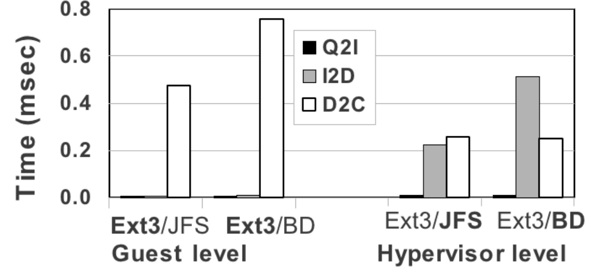


Figure 11: I/O times under sequential read workload.

can take advantage of the readahead at the hypervisor, achieving effective data caching. In contrast, when a disk partition is used, there is no readahead and data caching. Therefore, for all file systems, to gain high I/O performance, we recommend cloud administrators to select a flat file over raw partitions for services dominated by sequential reads.

### 4.3.2 Sequential Write Workload

Our investigation uncovers the root causes of the nested file systems' performance dependency under a sequential write workload in two cases: (A) two file system combinations hold the same host file system, and (B) two combinations hold the same guest file system. The analysis detailed below focuses on two principal factors: sensitivity of an I/O scheduler and effectiveness of block allocation mechanisms.

**A. Different guests (Ext3, JFS) on the same host (ReiserFS)** As shown in Figure 8 (B), we can see that the I/O performance of Ext3/ReiserFS is much worse than that of Ext3/BD, while the I/O performance of JFS/ReiserFS is much better than JFS/BD. At the guest level, we analyze the performance dependency of Ext3 and JFS based on the comparison of their I/O characteristics. The details of this comparison are shown in Figure 13.

Figure 13 (A) shows that most I/Os issued from Ext3 and sent to the block layer are well merged at the guest level's I/O scheduler. The effective merging of I/Os significantly reduces the number of I/Os to be served on Ext3 (guest). Meanwhile, Figure 13 (B) shows that 99% I/Os of Ext3 are in small size (8K) and those of JFS is 68%. Apparently, merging multiple small size I/Os incurs additional overhead. This is because the small requests have to be waited longer in the queue in order to be merged, thus, increasing their idle times. This behavior is illustrated in Figure 13 (C).

To understand the root cause of merging happened on Ext3 and JFS (guest), we perform a deep analysis by monitoring every issued I/O activities at the guest level.

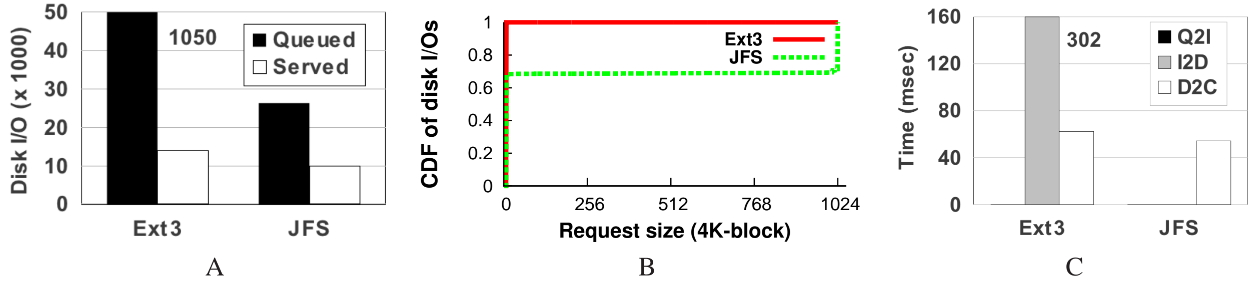


Figure 13: I/O characteristics at **guest level**: (A) disk I/Os, (B) I/O size, and (C) average I/O time.

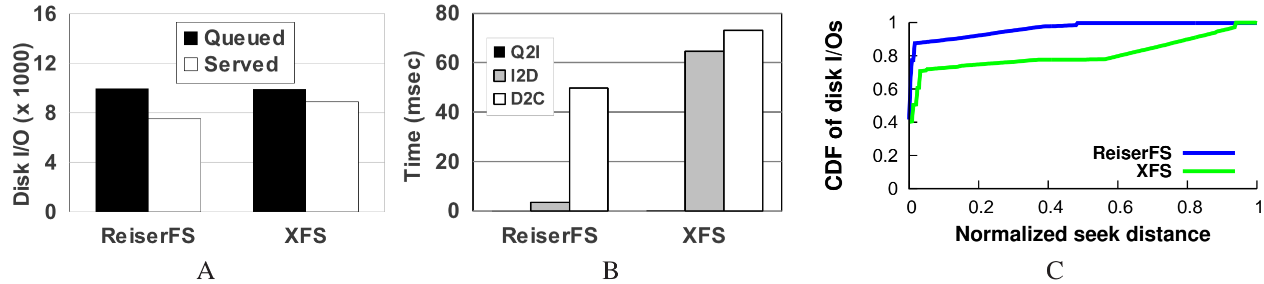


Figure 14: I/O characteristics at **hypervisor level**: (A) disk I/Os, (B) average I/O time, and (C) disk seeks.

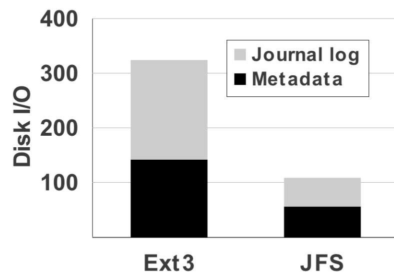


Figure 12: Extra I/O for journal log and metadata updates under sequential write workload.

What we found is that the block allocation mechanism causes this performance variation. To minimize disk seeks, Ext3 issues I/Os to allocate blocks of data on disk close to each other. The data includes regular data file, its metadata, and journal logs of metadata. This allocation scheme makes most I/Os be *back* merged. A back merge behavior denotes that a new request sequentially falls behind an exiting request on an order of the start sector, as they are logically adjacent. Note that two I/Os are logically adjacent when the end sector of one I/O is logically located next to the begin sector of the other I/O. As we can see, clustering adjacent I/Os facilitates the data access. However, it requires the issued I/Os to be waited longer in the queue for being processed.

JFS is more efficient than Ext3 in journaling. For regular data file written into disk, both Ext3 and JFS effectively coalescence multiple write operations to reduce the number of I/O committed into disk. However, for metadata and journal logs, instead of independently committing every single concurrent log entry as Ext3, JFS requires multiple concurrent log entries to be coalesced as one commit. For this reason, as shown in Figure 12, JFS has less I/Os spent for journaling, resulting in less performance degradation.

**Remarks:** The efficiency provided by the I/O scheduler's optimization is no longer valid for all nested file systems. Since file systems allocate blocks on disk differently, nested file systems have different impacts on performance when one particular I/O scheduler is used. Therefore, a nested file system should be chosen based on the effectiveness of underlying I/O scheduler's operations on its block allocation scheme.

**B. Same guest (JFS) on different hosts (ReiserFS, XFS)** Based on results of sequential writes shown in Figure 8 (B), JFS (guest) performs better on ReiserFS than on XFS. We analyze I/O activities of these host file systems to uncover differences of their block allocation mechanisms. The detailed analysis is given below.

The analysis of I/O activities reveals that the I/O scheduler processes ReiserFS' I/Os similarly to those of XFS. As shown in Figure 14 (A), the number of host file systems' I/Os to be queued and served are fairly similar in ReiserFS and XFS. However, Figure 14 (B) denotes that XFS' I/Os are executed slower than those of ReiserFS. A further analysis is needed to explain this behavior. In general, file systems allocate blocks on disk differently, thus, resulting in a different execution time for I/Os. For this reason, we perform an analysis on the disk seeks. Based on the results shown in Figure 14 (C), we find that long distance disk seeks on XFS cause high overhead and reduce its I/O performance. Note that in

Figure 15: Extra data written into disk under the same workload from JFS (**guest**).

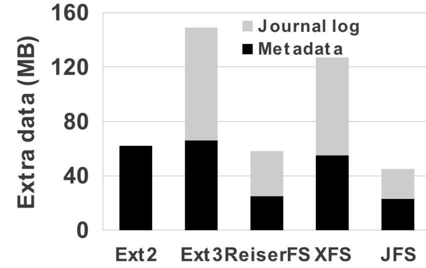


Figure 16: (**hypervisor level**) Extra data written into disk under a write-dominated workload from guest VM.

Figure 14 (C), the x-axis is represented as a normalized seek distance and 1 denotes the longest seek distance of the disk head, from one end to the other end of the partition.

With respect to the case of one host file system allocates disk blocks more effectively than another under the same workload, we analyze the mechanisms to allocate disk blocks of ReiserFS and XFS and find that XFS induces an overhead because of a multiple journal logging. The detailed explanations are as follows:

A multiple logging mechanism of metadata also incurs an overhead on XFS. Basically, XFS is able to record multiple separate changes occurred on the metadata of a single file and store them into journal logs. This technique effectively avoids such changes to be flushed into disk before another new change will be logged. However, every change of metadata can be range from 256 Bytes to 2 KB in size, while the default size of the log buffer is only 32 KB. Under an intensive write dominated workload, this small log buffer causes multiple changes of the file metadata to be frequently logged. As shown in Figure 15, this repeatedly logging produces extra data written into disk, thus, resulting in a performance loss.

**Remarks:** (1) An effective block allocation of one particular file system no longer guarantees a high performance when it runs on top of another file system. (2) Under an intensive write dominated workload, an update of journal logs on disk should be carefully considered to avoid performance degradation. Especially for XFS, the majority of its performance loss is attributed to not only a placement of journal logs, but also a technique to handle updates of these logs.

## 5 Discussion

Despite various practical benefits in using nested file systems in a virtualized environment, our experiments have shown the associated performance overhead to be significant if not configured properly. Here we offer five advice on choosing the *right* guest/host file system configurations to minimize performance degradation, or in some cases, even improve performance.

**Advice 1** For workloads that are read-dominated (both sequential and random), using nested file systems has minimal impact on I/O throughput, independent of guest and host file systems. For workloads that have a significant amount of sequential reads, nested file systems can even improve throughput due to the readahead mechanism at the host level.

**Advice 2** On the other hand, for workloads that are write-dominated, one should avoid using nested file systems in general due to i) one more layer to pass through and ii) additional metadata update operations. If one must use nested file systems, journaled file systems in the host should be avoided. Journaling of both metadata and data can cause significant performance degradation, and therefore, is not practical to use for most workloads, and if only metadata is journaled, a crash can corrupt a VM image file easily, thus, giving no benefit to metadata-only journaling mode in the host. As shown in Figure 16, the additional metadata writes to the journal log can result in significantly more I/O traffic. Performance is even more impacted if the location of the log is placed far away from either the metadata or the data locations.

**Advice 3** For workloads that are sensitive to I/O latency, one should also avoid using nested file systems. As shown in Figure 6, even in the best case scenarios, nested file systems could increase I/O latency by 10-30% due to having an additional layer of file system to traverse and one more I/O queue to wait for.

**Advice 4** In a nested file system, data and metadata placement decisions are made twice, first in the guest file system and then in the host file system. Guest file system uses various temporal and spatial heuristics to place related metadata and data blocks close to each other. However, when these placement decisions reach the host file system, it can no longer differentiate between data and metadata and treats everything as data. As a result, the secondary data placement decisions made by a host file system are both unnecessary and less efficient than those made by a guest file system. Ideally, the host file system should simply act as a pass-through layer such as VirtFS [22].

**Advice 5** In our experiments, we used the default set of formatting and mounting parameters in all the file systems. However, just like in a non-virtualized environment, these parameters can be tuned to improve performance. There are more benefits in tuning the host file system's parameters than guest's as it is ultimately the layer that communicates with the storage device.

One should tune its parameters in such a way that the host file system most resembles a "dumb" disk. For example, when a disk is instructed to read a small disk block, it will actually read the entire track or cylinder and keep them in its internal cache to minimize mechanical movement for future I/O requests. A host file system can emulate this behavior by using larger block sizes.

Metadata operations at host file system is another source of overhead. When a VM image file is accessed or modified, its metadata often has to be modified, thus, causing additional I/O load. Parameters such as *noatime* and *nodiratime* can be used to avoid updating the last access time without losing any useful information. However, when the image file is modified, there is no option to avoid updating the metadata. As the image file will stay constant in size and ownership, the only field in the metadata that needs to be updated is the last modified time, which for an image file is just pure overhead. Perhaps this can be implemented as a file system mount option. Note that journaling, as mentioned previously, in the metadata-only mode has very little usage in the host level.

Lastly, using more advanced file system features to configure block groups and B+ trees to perform intelligent data allocation and balancing tasks will most likely be counter-productive. This is because these features will cause guest file system's view of disk layout to deviate further from the reality.

## 6 Conclusion

Our main objective is to better understand performance implications when file systems are nested in a virtualized environment. The major finding is that the choice of nested file systems on both hypervisor and guest levels has a significant performance impact on I/O performance. Traditionally, a guest file system is chosen based on the anticipated workload, regardless of the host file system. By examining a large set of different combinations of host and guest file systems under various workloads, we have demonstrated the significant dependency of the two layers on performance, and hence, system administrators must be careful in choosing *both* file systems in order to reap the greatest benefit from virtualization. In particular, if workloads are sensitive to I/O latency, nested file systems should be avoided or host file systems should simply perform as a pass-through layer in certain cases.

The intricate interactions between host and guest file systems represent an exciting and challenging optimization space for improving I/O performance in virtualized environments. Our preliminary investigation on nested file systems will help researchers to better understand critical performance issues in this area, and shed light on finding more efficient methods in utilizing virtual storage. We hope that our work will motivate system designers to more carefully analyze the performance gap at the real and virtual boundaries.

## Acknowledgements

We are grateful to the anonymous referees and our shepherd, Andrea Arpaci-Dusseau, for their detailed feedback and guidance. This work was partially supported by NSF grant 0901537 and ARO grant W911NF-11-1-0149.

## References

[1] Amazon Elastic Compute Cloud - EC2. `http://aws.amazon.com/ec2/` [Accessed: Sep 2011].

[2] blktrace - generate traces of the I/O traffic on block devices. `git://git.kernel.org/pub/scm/linux/kernel/git/axboe/blktrace.gitbt` [Accessed: Sep 2011].

[3] Filebench. `www.solarisinternals.com/wiki/index.php/FileBench` [Accessed: Sep 2011].

[4] FIO - Flexible I/O Tester. `http://freshmeat.net/projects/fio` [Accessed: Sep 2011].

[5] IBM Ccloud Computing. `http://www.ibm.com/ibm/cloud/` [Accessed: Sep 2011].

[6] Nested svm virtualization for kvm. `http://avikivity.blogspot.com/2008/09/nested-svm-virtualization-for-kvm.html` [Accessed: Sep 2011].

[7] The QCOW2 Image Format. `http://people.gnome.org/~markmc/qcow-image-format.html` [Accessed: Sep 2011].

[8] VirtualBox VDI. `http://forums.virtualbox.org/viewtopic.php?t=8046` [Accessed: Sep 2011].

[9] VMware Tools for Linux Guests. `http://www.vmware.com/support/ws5/doc/ws_newguest_tools_linux.html` [Accessed: Sep 2011].

[10] VMWare Virtual Disk Format 1.1. `http://www.vmware.com/technical-resources/interfaces/vmdk.html` [Accessed: Sep 2011].

[11] Window Azure - Microsoft's Cloud Services Platform. `http://www.microsoft.com/windowsazure/` [Accessed: Sep 2011].

[12] Xen Hypervisor Source. `http://xen.org/products/xen_archives.html` [Accessed: Sep 2011].

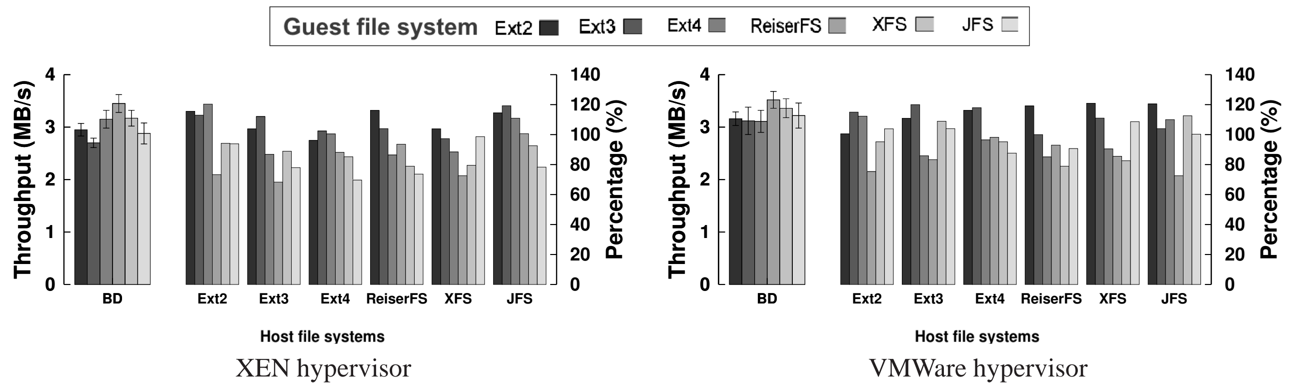


Figure 17: Other hypervisors show variation of relative **I/O throughput** of guest file systems under database workload (**higher is better**)


[13] Xen source - progressive paravirtulization. `http://xen.org/files/summit_3/xen-pv-drivers.pdf` [Accessed: Sep 2011].

[14] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX ATC'05*, April 2005.

[15] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *USENIX OSDI'10*, October 2010.

[16] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *USENIX FAST'09*, February 2009.

[17] D. Boutcher and A. Chandra. Does virtualization make disk scheduling passé? In *USENIX HotStorage'09*, October 2009.

[18] L. Cherkasova, D. Gupta, and A. Vahdat. When virtual is harder than real: Resource allocation challenges in virtual machine based IT environments, Feburary 2007.

[19] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (bvt) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, SOSP '99, Charleston, SC, USA, 1999.

[20] H. Huang, W. Hung, and K. G. Shin. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, Brighton, United Kingdom, 2005.

[21] K. Huynh and S. Hajnoczi. KVM/QEMU Storage Stack Performance Discussion. In *Proposals of Linux Plumbers Conference*, Cambridge, MA, USA, November 2010.

[22] V. Jujjuri, E. V. Hensbergen, and A. Liguori. VirtFS - A virtualization aware File System pass-through. In *Proceedings of the Ottawa Linux Symposium*, 2010.

[23] M. Kesavan, A. Gavrilovska, and K. Schwan. On Disk I/O Scheduling in Virtual Machines. In *USENIX WIOV'10*, Pittsburgh, PA, USA, March 2010.

[24] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, 2007.

[25] D. Ongaro, A. L. Cox, and S. Rixner. Scheduling I/O in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, Seattle, WA, USA, 2008.

[26] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[27] S. R. Seelam and P. J. Teller. Virtual I/O scheduler: a scheduler of schedulers for performance virtualization. In *ACM VEE'07*, June 2007.

[28] P. J. Shenoy and H. M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In *Proceedings of ACM SIGMETRICS Conference*, 1997.

[29] S. Sivathanu, L. Liu, M. Yiduo, and X. Pu. Storage Management in Virtualized Cloud Environment. *IEEE Cloud Computing'10*, 2010.

[30] K. Suzaki, T. Yagi, K. Iijima, N. A. Quynh, and Y. Watanabe. Effect of readahead and file system block reallocation for lbcas. In *Proceedings of the Linux Symposium*, July 2009.

[31] C. Tang. Fvd: a high-performance virtual machine image format for cloud. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, Portland, OR, 2011.

[32] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. On the DMA mapping problem in direct device assignment. In *SYSTOR'10: The 3rd Annual Haifa Experimental Systems Conference*, Haifa, Israel, May 2010.


## Appendix

We have conducted experiments with the database workload to verify if the I/O performance of nested file systems is hypervisor-dependent. The chosen hypervisors are architecturally akin to KVM, such as VMware Player 3.1.4 with guest tools [9], and Xen 4.0 with Xen paravirtualized device drivers [12]. Figure 17 shows that the I/O performance variations of guest file systems on Xen and VMware are fairly similar to those on KVM.