Storage Deduplication and Management for Application Testing over a Virtual Network Testbed

Chang-Han Jong[1], Pin-Jung Chiang[2], Taichuan Lu[3], and Cho-Yu Chiang[3]

[1] University of Maryland, College Park, MD
chjong@umd.edu
[2] National Taiwan University and Chunghwa Telecommunication Laboratories, Taiwan
brchian@cht.com.tw
[3] Telcordia Technologies, Piscataway, NJ
{tedlu,chiang}@research.telcordia.com

Abstract. With the virtual machine technologies, Virtual Ad hoc Network (VAN) testbed was designed to evaluate functional correctness and communication performance of Mobile Ad hoc Network (MANET) applications. When VAN is used for large-scale testing that requires hundreds of virtual machines, storage redundancy becomes an issue. Although Content Addressable Storage (CAS) techniques were designed to address the storage redundancy issue, it incurred online hash computation overhead for every write access to disk blocks, which affects testing accuracy. We present File-level Block Sharing (FBS) that achieves the same functionality of CAS while removing the online computation overhead. By getting file-to-block mappings through read-only mounting, FBS only needs to handle the blocks belonging to newly-installed files offline and thus incurs little online overhead. Our prototype showed no online overhead statistically and low offline overhead. The prototype was developed and its overhead with respect to block-level storage deduplication was analyzed under both Ext2/3/4 and NTFS file systems.

Keywords: testbed, MANET, VAN, FBS, deduplication.

1 Introduction

The dynamic nature of Mobile Ad hoc Network (MANET) makes application testing a grand challenge. Node mobility, intermittent link connectivity and multi-hop wireless communication interference in MANETs cannot be easily fabricated in a testbed environment. The other critical requirement for MANET application testing is that source code modification needs be avoided: testing abstract models of actual applications loses fidelity while using different versions of software for lab testing and for field testing causes serious software consistency maintenance issues.

Virtual Ad hoc Network (VAN) testbed [1] was designed to allow unmodified applications to communicate over a simulated MANET. It was designed to evaluate functional correctness and communication performance of MANET protocols and applications. VAN testbed supplies a testing environment in which unmodified applications can send packets over a virtual network realized by high-fidelity simulation [11]. VAN testbed uses virtual machines to host applications, thereby simplifying testbed management and reducing hardware requirements.

VAN testbed achieves the goal of requiring neither code change nor special environment configuration by leveraging host virtualization technologies and VLAN configuration at the virtual machine monitor layer [1]. For each mobile node under test, application software can be installed in a dedicated copy of operating system environment. Although this approach simplifies testing environment setup and achieves high testing fidelity, it imposes a significant storage requirement when large-scale testing needs to be performed on a VAN testbed. For example, a typical Linux installation for application testing takes roughly 10GB of disk space. For a 100 node testing scenario, it requires 1 TB disk space. Since VAN testbed was conceived to allow time-shared access by multiple users with different testing scenarios, to support ten different 100-node testing scenarios, 10 TB of storage space is required. Note that a significant amount of storage is used by duplicated data, as typically most of the nodes under test share similar host environment configurations. Therefore, even though RAID [6] is being used in VAN testbed, considering the cost of RAID, the storage deduplication issue needs be addressed in order to scale up the testbed.

In light of host virtualization, from a guest operation system's perspective, storage deduplication approaches can be classified into two categories: file-level and block-level. File-level deduplication approaches hinge on identifying files that can be shared [2]. Approaches in this category include mounting read-only shared folders by Network File System (NFS) [12][13], using the Copy-on-Write (CoW) technique by Union File System [7], as well as creating symbolic links for common files. The main issues associated with file-level storage deduplication approaches are i) it requires human configuration to identify and take care of the common files; and ii) it lacks flexibility to deal with ever-changing disk access needs; iii) NFS performance is much slower than block-level storage[12][14]. Therefore, these approaches are not desirable when the testing environment needs to be updated from time to time, which is a common practice during test. In contrast, block-level storage deduplication approaches are typically agnostic of file systems. Their main advantage is that the deduplication process can be fully automated without requiring human in the loop. Certain storage virtualization technologies, such as Logical Volume Manager (LVM), can provide virtual disks to virtual machines and employ copy-on-write technique to maximize block sharing [6]. When a virtual machine (VM) is started, based on its configuration a duplicate of a template disk image stored on the block storage device can be instantiated within a second. The duplicate, namely *snapshot*, is a virtual disk mapped to the template image by default and used only to store and serve modified blocks to save block device storage space. All read access to blocks that have never been modified will be retrieved from the template disk image on the block storage device.

One limitation of snapshot-based virtual disks is that after the snapshots have been instantiated, modified blocks with identical contents will not be shared. We name this problem as *post-snapshot block sharing problem*. Content Addressable Storage (CAS) [3][5] was introduced to tackle this problem by computing the hash values for each block stored in the snapshot and coalescing the blocks that share the same hash values. However, online hash computation overhead becomes a serious issue for testing on a VAN testbed, especially when large-scale application testing involves substantial disk write operations (e.g., logging) from the nodes under test.

In this paper, we present File-Level Block Sharing (FBS) to address the post-snapshot block sharing problem. One salient feature is that FBS does not incur the run-time overhead associated with CAS. In a nutshell, FBS checks whether a target block already has a duplicate in the other snapshots by checking only their blocks belonging to the file that has the same filename as the target block. FBS achieves storage deduplication by taking advantage of file-to-block mappings and thus avoid online hash calculation for each write operation. FBS can be used without modifying guest operating systems, neither does it require running another utility program inside virtual machines. As a proof-of-concept exercise, we have implemented a prototype using VirtualBox [9] for virtual machine environment, Linux 2.6 kernel for the operating system, LVM for block-level storage device management, and iSCSI protocol [14] for transferring data between the block storage device and the snapshot virtual disk.

The rest of the paper is organized as follows. Section 2 provides background information for this research work by briefly introducing related work. Section 3 presents the File-Level Block Sharing approach. Section 4 describes the implementation prototype. Section 5 provides performance analysis and evaluation of the prototype. We conclude in Section 6 this paper and point out possible future work.

2 Related Work

In a VAN testbed, test scenarios could consist of hundreds of nodes and durations of scenarios could range from a few minutes to a couple of days. VAN testbed uses virtual machines to support its operations. As the hypervisor underlying virtual machines is Linux-based, LVM was selected to manage logical disk volumes and mass-storage devices such as RAIDs. The term "*volume*" refers to a disk drive or a partition thereof. LVM can be regarded as a thin software layer underneath the operating system to provide virtual disks and manage hard disks and their partitions. This layer of abstraction provides ease-of-use in managing actual disk drives, including creating snapshot virtual disks from a template disk image. Snapshots are critical for establishing, maintaining, and managing a virtual network testbed consisting of hundred of nodes.

Even though the snapshot functionality facilitates rapid replication of virtual disks (a.k.a. logical volumes), nodes on the testbed cannot use virtual disks with identical contents for many reasons. For example, each node is supposed to have a unique node name, a unique IP address, a unique MAC address, etc. In addition, during the testing process applications and OS need to write contents to disks. Moreover, application testing is an incremental process—typically the needed packages and software updates take place frequently after setting up a common template image. This is referred to as post-snapshot block sharing problem, as the many copies of snapshots will accumulate a considerable amount of identical contents over the course of testing. For example, we have seen a scenario with roughly 100 nodes having 3G bytes of almost identical data in each snapshot. Content Addressable Storage (CAS) tackles this problem in a two-step approach. First, hash values of every block written to the storage are computed and stored. Second, for blocks having the same hash values, only one copy is kept. Access to the shared blocks will be through indirection. We discuss two different implementations of the CAS approach below.

IBM's Duplicate Data Elimination (DDE) [3] was the first CAS implementation running on IBM's cluster file system, Storage Tank [8]. Storage Tank is composed of clients, meta-data servers and Storage Area Network (SAN) devices [15]. Client computers interact with meta-data servers to lock/unlock files and obtain block allocations. Client computers can then directly access SAN without any meta-data servers in the data path. For each block that needs to be modified, client computers calculate SHA-1 hash values for the block and return them to the meta-data server, which stores the hash values of the modified blocks. A particular meta-data server is responsible for coalescing the blocks with identical hash values.

To alleviate the performance issue resulting from using a single meta-data server to coalesce blocks, VMWare designed and implemented a decentralized storage deduplication (DeDe) scheme [5] for its VMFS [16]. The aim was to distribute the workload of detecting duplicate blocks to client computers as VMFS does not use a central meta-data server. According to [5], online hash computation consumes a lot of CPU cycles and therefore one CPU core on a blade is dedicated to hash value computation. No matter whether one CPU core is dedicated for hashing or not, the testbed is affected by either the degraded accuracy or increased computation cost.

3 File-Level Block Sharing

The goal of File-Level Block Sharing (FBS) is to achieve post-snapshot block sharing with less online computation overhead. The basic idea is to associate file-level semantics with blocks on block-level storage devices. Common files across snapshots for virtual machines are collected and stored in a volume managed by LVM. Access to these common files will be redirected by the virtual disk drivers to this volume.

3.1 File-Level Semantics in Block-Level Storages

The relation between a file and its blocks is maintained by the file system. When an application accesses a file through the file system, the file system retrieves the meta-data of the file to obtain the logical addresses of the disk blocks that belong to the file. On the other hand, FBS needs to obtain the mappings from blocks to the files owning the blocks in order to perform block deduplication. To discover the mapping from a block to the file using the block is difficult because the mapping from a file to its blocks is indexed by the file system. Although mapping from a block to the file by snooping and parse all I/O access, the formats of the file-to-blocks meta-data used by the various file systems are different. Fortunately, when developing a file system, debugging utilities or libraries are also made available as by-products. They can be used to read file meta-data in user land. For example, NTFS has *libntfs* library and Linux ext2/ext3/ext4 has *e2fsprogs* utility. FBS uses these tools/libraries to retrieve file-to-block mappings and thus derive block-to-file mappings.

3.2 Algorithm

By using LVM, a block-level storage device uses one template volume and *k* copy-on-write volumes to implement *k* virtual disks for *k* virtual machines, respectively. FBS requires use of an additional common volume to store the blocks shared by the virtual disks. Typically this is for the VAN testbed to maintain a common volume that contains all add-on packages needed for testing scenarios after the template volume has been made.

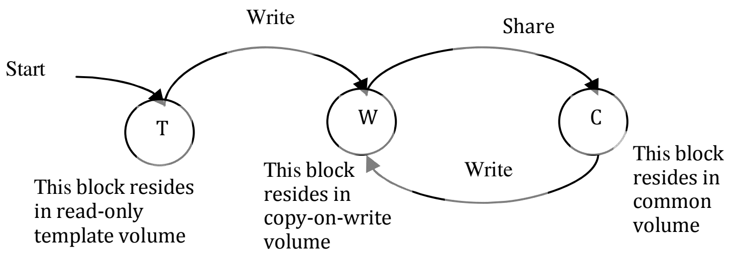

Start
Write
Share
T
W
C
Write
This block resides in read-only template volume
This block resides in copy-on-write volume
This block resides in common volume

Fig. 1. FBS state transition diagram of a single block

A virtual disk driver contains a block mapping data structure to map the read/write access of a virtual disk to template volume, to the per-virtual-disk copy-on-write volume, and to the common volume. Initially when the snapshot virtual disk is created, all block accesses are directed to the template volume. If a block is modified, the access to it will be directed to the copy-on-write volume belonging to the virtual disk. If this block is to be shared after it has been modified, it will reside in the common volume. Fig. 1 illustrates the state transition diagram of a single block.

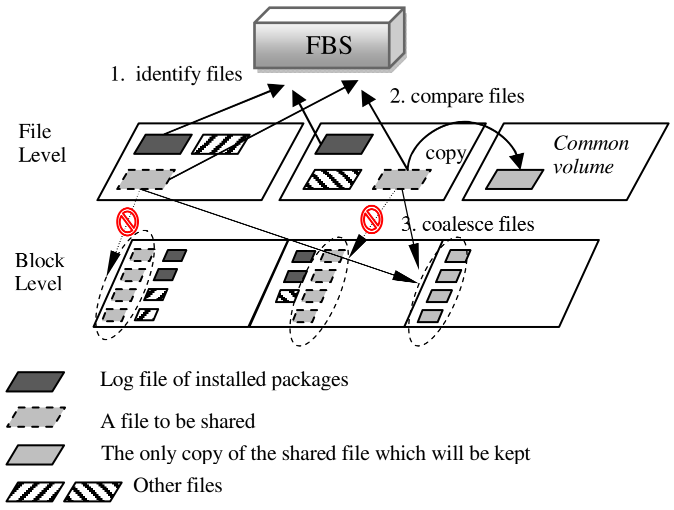


Fig. 2. High-level operational view of FBS algorithm

The FBS algorithm is listed below and also illustrated in Fig. 2.

1. **Identify files:** FBS identifies which files across virtual disks are likely to share the same contents.
2. **Compare files:** For each identified file, FBS checks if it is already shared in the common volume. If so, perform the next step after verifying its content is the same as the block in the common volume. Otherwise, if two or more identical files have been compared but without a match in the common volume yet, store a copy of the file in the common volume. The content verification and the copying action are performed at the file level.
3. **Coalesce blocks:** Access to the blocks of shared files will be directed to the blocks in the common volume.

In the figure we can see that there are three volumes at the file-level. The left two are the virtual disk volumes and the right one is the common volume. In the first step, FBS use the package installation log to identify the files which are likely to be shared. Then FBS compare these files in different volumes. FBS may decide to copy the to-be-shared files to the common volume. In the final step, FBS checks the file-to-block mapping for these to-be-shared files and inform the storage to modify the access mapping. The first two steps are file-based operations. The third step requires updates to the file-to-blocks mappings in the storage system. Implementation details will be discussed in the next section

4 Prototype

FBS prototype was developed using the following setting. Multiple Linux host machines run VirtualBox virtual machine monitor and use iSCSI protocol to access a volume on a Storage Area Network (SAN) storage device. Host machines share a global view of the storage volume and use Linux LVM to create virtual disks. The LVM partitions the volume(s) provided by SAN to multiple small volumes as virtual disks.

The host operating system underneath the virtual machines is 64bit Ubuntu Linux Server 10.04 using a custom LVM with our own patch. Operating systems for the VirtualBox virtual machines are 32bit Ubuntu Linux Server 10.04 that uses ext4 file system with 4Kbytes block size. LVM was also configured to use 4Kbytes chunk size for snapshot virtual disks.

4.1 Software Components

FBS uses three software components to implement its algorithm. We implemented *File System Agent* and *Storage Agent* by python/C, while *Storage* was coded by modifying the source code of LVM. These three components and their relationships are shown in Fig. 3.

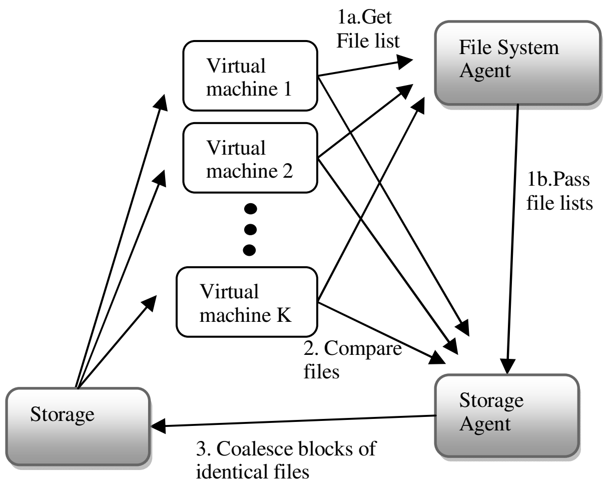


Fig. 3. Logical View of FBS Software Components

File System Agent (FA) performs the first step of the algorithm to identify files with identical contents. FA first mounts the virtual disk in read-only mode. Read-only mounting prevents virtual machines from being shutting down. Then, FA reads the database of the software package management system that is used by Ubuntu. By the information stored in the database, FA generates a file list which includes files installed after the snapshot virtual disks have been made. Since FBS was designed for the VAN testbed, we are concerned of only the newly installed software that results in storage redundancy. As a side note, if FBS is used outside of the scope of VAN, FA can do a complete file system sweep to build a list of all the files modified/created after the snapshot virtual disks have been made by using *find* command.

Storage Agent (SA) performs the second step of the algorithm, primarily to manage the common volume. For each file that could have duplicates, SA checks whether a duplicate of the file exists in the common volume. The common volume is formatted in ext4, same as the virtual disks. If a file currently stored in the copy-on-write volume already has an identical copy in the common volume, SA uses Linux *debugfs* utility to retrieve the block allocation of the file and then informs *Storage* to modify the mappings to point to the blocks of the shared copy in the common volume. If a file doesn't have a copy already stored in the common volume but multiple files on different virtual disks share the same content, SA copies the file to the common volume and then informs *Storage* to modify the mappings.

Storage (ST) manages the virtual disks and performs the last step of the algorithm. ST provides an ioctl interface, which is used to control the block mappings of the virtual disks.

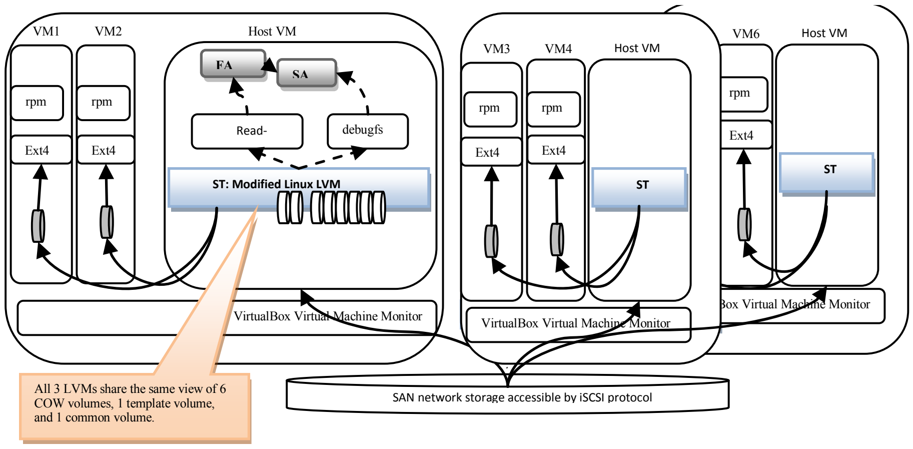

VM1
VM2
Host VM
FA
SA
rpm
Ext4
Read-
debugfs
ST: Modified Linux LVM
VirtualBox Virtual Machine Monitor
VM3
VM4
ST
VM6
ST
All 3 LVMs share the same view of 6 COW volumes, 1 template volume, and 1 common volume.
SAN network storage accessible by iSCSI protocol

Fig. 4. A FBS prototype implementation with six virtual machines

To put everything in context, we explain the system architecture and setup in a scenario consisting of six virtual machines as shown in Fig. 4. FA and SA reside in the host VM on the left. The storage components, i.e., the modified LVM, have instances in every machine and they share the same global view of virtual disks. In other words, each machine can see all the virtual disks available on the SAN storage. FA mounts virtual disks in read-only mode so that FA would not affect the 6 virtual machines (VM1,VM2,…, VM6). FA reads the installation logs of the package management system and passes the newly installed files list to SA. SA then compares files and identifies those to be shared and put their copies in the common volume. For all files sharing a copy in common volume, SA invokes *debugfs* to get the block allocation of each file and informs ST to remap the block access from the original virtual disk to the common volume for the newly shared file. In this 6 virtual machine example, ST maintains totally eight volumes: one for the template volume, one for the common volume, and the other six for the copy-on-write volumes.

4.2 FBS vs. CAS

Since FBS was designed to address the storage redundancy issue for a virtual network testbed rather than for host virtualization in general, it is done differently than most other block-level storage deduplication approaches in many ways. As an example, we compare FBS against CAS and show their major differences in Table 1.

Table 1. Comparing CAS and FBS

	CAS	**FBS**
Sharing unit	a block in a virtual disk	a file in a virtual disk
Non-sharable blocks	file system meta- data	file system meta-data; files which have some identical blocks but have at least one non-identical block
Back-end storage	Cluster file system	LVM
Volumes needed for *k* virtual disks	1 template volume + k copy-on-write volumes	1 template volume + k copy-on-write volumes + 1 common volume
Major online overhead	Hashing of written blocks	None
Major offline overhead	Coalescing blocks	Comparing files with files in common volume+Coalescing blocks of files

Sharing. Block-level deduplication approach such as CAS uses a block as its sharing unit. Since FBS associates blocks with the files they belong to, we use file as the sharing unit. More specifically, FBS considers all blocks pertaining to a file in a virtual disk image as the sharing unit. In most cases, the files that FBS processes are much smaller than the files in CAS' cluster file system because the files in CAS for virtual machine environment are virtual disk images while the files in FBS are the files as seen by operating system instances across the testbed. Using file as the sharing unit also has the drawback that two file are not sharable if they are not completely identical in all blocks. However, in VAN storage redundancy is mainly consequences of software installation, we believe using file as the sharing unit best suits our needs.

Storage. Although ST currently is implemented as a block-level storage, ST could be implemented by a cluster file system as well. We chose LVM mainly because VAN already used LVM and we did not want to change the storage setting. In addition, while CAS uses cluster file system to provide virtual disk images, FBS prototype directly provides virtual disks from SAN. An indicative, but inaccurate way to calculate storage efficiency is by the total number of volumes used. If using CAS, k snapshots of a template virtual disk require 1 template volume and k copy-on-write volumes; FBS needs all the above and an additional common volume. The common volume provides the benefit that the VAN users can setup a common volume with all the files to be shared.

Overhead. FBS has lower hashing overhead than CAS. For online overhead, FBS does not have the online hashing overhead at the block-level as CAS has. On the other hand, FBS does calculate hashing values when offline at the file-level while CAS does not. However, the overhead of calculating hash at the file-level is lower than at the block-level due to the caching/perfecting mechanism and the reduced amortized overhead. For example, computing MD5 hash by OpenSSL library need to call MD5_init() for initialization, MD5_Update() for computing for each 512-bit chunk of data, and MD5_Final() to generate the final hash output [10]. If hashing is computed on a file, instead of a single block, the overhead of MD5_Init() and MD5_Final() will be amortized.

5 Evaluation

We evaluated both offline and online overhead of FBS, and the associated storage efficiency of the file systems. The offline overhead is due to the file hashing operations performed by SA. The online overhead is due to the newly-introduced table of block mappings of virtual disks. The efficiency of FBS can be affected by the file size and the layout of file system running above. We will discuss efficiency with respect to two famous families of file systems, ext2/3/4 and NTFS.

Conceptually, data are sharable but meta-data are not. The meta-data of a file stores the pointers to the data blocks and other information. Since the layout of the file system depends on the order of files written to a file system, the pointers to the data blocks of the same file in different file systems will likely have different values. Therefore, meta-data is not sharable.

5.1 Offline and Online Overhead

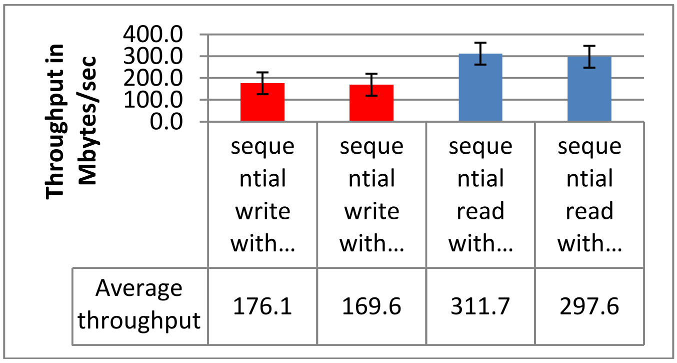
The first set of experiments was to evaluate the offline overhead of FBS, namely, the time SA took to traverse a file system and compare files via hashing. Based on the VAN testbed scenario described in the previous section, we set up an evaluation environment that used 42 VirtualBox virtual machines. The virtual disks for the virtual machines were snapshots of a template volume that has Ubuntu Linux Server 10.04 installed in it. We installed openoffice.org suite and all depended-on packages on all virtual disks. The total installed file size was 438MB. Openoffice.org was chosen for the study simply because of its large size and popularity.

Table 2. FBS offline overhead

	Mean	**Std dev**
Total Processing Time for 42 Virtual Machines	677.09 sec	8.14 sec
Average Processing Time for 1 Virtual Machine	677.09/42=15.79 sec	N/A

Table 2 shows the time needed for FBS to perform offline processing. To process all 42 virtual disks FBS spent 677.09 seconds on average of five runs. Since SA performed the operation in serial, to process one virtual disk would need roughly 15.79 seconds. The average throughput was 27.74 megabytes per second.

The second set of experiments was to evaluate the online overhead of FBS, namely, the throughput degradation of the storage system. We use Bonnie++ benchmark to evaluate the reading and writing throughput [4]. Bonnie++ is chosen because it performs intensive sequential reading and writing blocks to test the block-level storage performance. According to Fig. 5, we can see that after FBS was introduced, the average throughput in both sequential read and sequential write is almost the same, actually even decreased. Though it is possible that FBS increases the cache performance and therefore FBS has even a slightly better performance. However, after running the statistical t-test, we found that with or without FBS, the average throughputs



	sequential write with...	sequential write with...	sequential read with...	sequential read with...
Average throughput	176.1	169.6	311.7	297.6

Fig. 5. FBS online overhead

in sequential read are the same statistically (P Value=0.92). The average throughputs in sequential write are also the same statistically (P Value=0.82). In brief, the online performance of FBS and original LVM are the same in the statistical sense.

5.2 Linux Ext2/3/4 File System

Linux has its native file systems, Ext2, Ext3 and Ext4. They are mostly backward compatible. Ext3 adds journaling and Ext4 extends the support of large files, including the *Extent* feature. If the *extent* feature is disabled, file meta-data across the three systems would be identical.

Each file in the file system has an inode entry, possibly composed of a few indirect blocks along with data blocks. Each inode entry is 128 bytes and contains the pointers to the data block. When the embedded pointers are not enough, indirect blocks are allocates to store additional pointers to data blocks. Multiple inodes are squeezed into one block to save space. For example, 32 inodes can be fit into a 4KBytes block. Since two identical files residing in different file systems most likely will be assigned to data blocks in the different logical address, sharing inode or indirect blocks which contain the pointers to the data blocks are meaningless. On the other hand, due to the fact that Ext file system family uses a de-coupled approach to put inodes and data blocks in different areas, data blocks are shareable even if corresponding inodes are not sharable.

The following list shows an example of the block allocation for /bin/gzip. The file system contains a good amount of meta-data including file type, block allocation, access authorization, latest access time, etc. The information we are interested is block allocation. This file contains totally 15 blocks. In additional to the inode, gzip has 14 data blocks and one indirect inode block (block #854520). The inode of ext2/3 file systems has 12 pointers to point to 12 data blocks. If the file size is large than 12 blocks, the file system will allocate another indirect block to store data block pointers. In this example, /bin/gzip has more than 12 blocks, so one indirect block (block #854520) is used to store the 3 remaining data blocks pointers (block #854521~#854522). It is notable that indirect blocks introduce data block fragmentation. In this case the indirect block is between block #854519 and block #854521.

As Ext2/3 file systems were designed to support large files, multi-level indirect blocks are used. The first level is called simple indirect blocks and double/triple-indirect blocks are for the second and third levels. We summarize the formulae for meta-data sizes for Ext2/3 file system in Table 3. Also, Fig. 6 shows the meta-data overhead except for extremely large files. The meta-data overhead is quite low (1~2 indirect blocks) when the file size is not larger than 1037 blocks, which is around 4Mbytes. For the packages we installed in the evaluation, we found that much less than 1% of the files are big files that exceed 1036 blocks.

List 1. Ext2 Meta-data of the file /bin/gzip

```
Inode: 210540   Type: regular    Mode:  0755   Flags: 0x0   Generation: 3290196032
User:     0   Group:     0   Size: 53488
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 120
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x4ad48571 -- Tue Oct 13 09:49:37 2009
atime: 0x4af4e498 -- Fri Nov  6 22:08:08 2009
mtime: 0x473c3258 -- Thu Nov 15 06:49:44 2007
BLOCKS:
(0-11):854508-854519, (IND):854520, (12-13):854521-854522
TOTAL: 15
```

Table 3. Meta-data Overhead in Ext2/Ext3/Ext4 without Extents

X: # of blocks	**Meta-data**	**Meta-data type**
1~12 blocks	128Bytes	Inode and Simple Direct blocks
13~1036 blocks	128Bytes + 4Kbytes	Inode, Direct blocks and Simple Indirect blocks
1037~1049612 blocks	128Bytes + 4Kbytes*(2+Ceiling(X-1036) /1024)	Inode, Direct blocks, Simple Indirect blocks, and Double-indirect blocks
1049613~ 1074791436 blocks	128Bytes + 4Kbytes(1026+Ceiling(X-1049612)/1024^2+Ceiling(X-1049612)/1024)	Inode, Direct blocks, Simple Indirect blocks, Double-indirect blocks, and Triple-indirect blocks

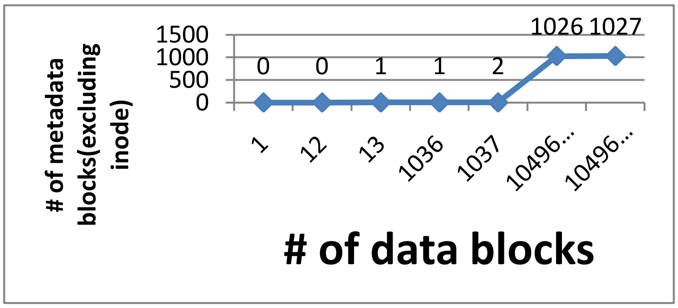


Fig. 6. Ext2/3/4 File system meta-data overhead

The Extent feature is an approach to reducing the file system overhead for large files. It has been used commonly in the modern file systems. The conventional way to represent the block allocation of a file is using a list of block addresses. The main drawback of this approach is that deleting a large file is very time consuming and the meta-data of large files would require considerable disk space. An extent contains the logical block address from the beginning of the file, beginning of the data block on the disk, and the number of consecutive data blocks. If a file contains only one set of consecutive data blocks, one extent entry would be enough to represent the file regardless of its size. Table 4 shows the meta-data overhead of Ext 4 file system when extents are used. An inode has totally 4 extents. Each extent maps a range of logical address to another range of physical address on the disk. So a very large file (e.g. DVD image) does not additional meta-data block other than inode if the data blocks of the file consists of 4 consecutive block sets or less. If a file has more than 4 consecutive block sets, indirect blocks are required and each indirect block can store up to 340 extents for 4Kbytes block size. To take advantage of extents, some file systems, like Ext4, use delayed I/O to make allocated blocks consecutive if possible.

Table 4. Meta-data Overhead in Ext4 with Extents

Y: # of consecutive blocks sets	**Meta-data size**	**Meta-data type**
4	128Bytes	Inode
>4	128Bytes +4K(1+Ceiling(Y-4)/340))	inode, index node, and leaf nodes

FBS can benefit from file systems supporting extents. If a file occupies consecutive blocks and can be represented by only one extent, changing the block-allocation mapping only needs one write operation.

5.3 Windows NTFS File System

Since NTFS is not an open standard, the following description of NTFS is empirical-based. For a disk volume, NTFS pre-allocate an area named Master File Table (MFT) to store the metadata. MFT contains 1 KBytes meta-data records and 12.5% of the volume is reserved for MTF. Each record contains the file information, including extents. A MFT record may contain the file itself if the file is small enough to fit to the record, say 700~800 Bytes. This kind of files cannot apply any block-level deduplication. For a file which cannot be squeezed in a MFT record, the meta-data size is 1Kbytes for one MFT record if the file has no more than 30 consecutive blocks sets. If a file has more than 30 consecutive sets of block sets, additional meta-data is required. Table 5 summaries the meta-data overhead of NTFS file system.

Table 5. Meta-data Overhead in NTFS

Z: # of consecutive blocks sets	meta-data size	Meta-type type
File size<P, P=700~800	0	Embedded
<=30 sets of consecutive blocks	1Kbytes record	MFT
>30 sets of consecutive blocks	1Kbytes*(ceiling(Z/30))	MFT

6 Conclusion and Future Work

The File-level Block Sharing (FBS) approach presented in this paper addresses the post-snapshot block sharing problem associated with reducing storage redundancy for virtual network testbed environments. Unlike other block-level storage approaches like CAS, FBS does not perform online hashing computation for every written blocks and therefore FBS is suitable for use in a virtual network environment such as Virtual Ad Hoc Network Testbed (VAN). FBS associates file-level semantics with block-level devices by using the file-to-block mapping obtained by the file system debugging tools/library. We have implemented a prototype based on Linux LVM/iSCSI and it shows no performance degradation in the statistical sense with respect to I/O throughput. On analyzing the ext2/3/4 and NTFS file systems, we discovered that the space efficiency of FBS was affected by the meta-data layout of file systems.

To improve offline processing time, we are thinking of implementing a software installation operation without actually copying the files. It may be achieved by modifying the package management software to install a software package by modifying the file system meta-data and directly use FBS to map the corresponding file access to the common volume.

References

[1] Poylisher, A., Serban, C., Lee, J., Lu, T.-C., Chadha, R., Chiang, C.-Y.J., Orlando, R., Jakubowski, K.: A Virtual Ad hoc Network Testbed. International Journal of Communicaiton Networks and Distributed Systems 5(1/2), 5–24 (2010)

[2] Pfaff, B., Garfinkel, T., Rosenblum, M.: Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In: 3rd Symposium on Networked Systems Design and Implementation, San Jose, California, pp. 353–366 (2006)

[3] Hong, B., Plantenberg, D., Long, D.D.E., Sivan-Zimet, M.: Duplicate Data Elimination in a SAN File System. In: 21st IEEE/12th NASA Goddard Symposium on Mass Storage Systems, Adelphi, Maryland, pp. 101–114 (2004)

[4] Bonnie++ benchmark suite, `http://www.coker.com.au/bonnie++/`

[5] Clements, A.T., Ahmad, I., Vilayannur, M., Li, J.: Decentralized Deduplication in SAN Cluster File Systems. In: USENIX Annual Technical Conference, San Diego, California (2009)

[6] Teigland, D.: Volume Managers in Linux. In: USENIX Annual Technical Conference, Boston, Massachusetts, pp. 185–198 (2001)

[7] Quigley, D., Sipek, J., Wright, C.P., Zadok, E.: Unionfs: User- and Community-Oriented Development of a Unification File System. In: Linux Symposium, Ottawa, Canada, pp. 349–362 (2006)

[8] Menon, J., Pease, D.A., Rees, R., Dulanovich, L., Hillsburg, B.: IBM Storage Tank-A Heterogeneous Scalable SAN File System. IBM Systems Journal 42(2), 250–267 (2003)

[9] Oracle VirtualBox, `http://www.virtualbox.org/`

[10] OpenSSL cryptography library, `http://www.openssl.org/`

[11] Biswas, P.K., Serban, C., Poylisher, A., Lee, J., Mau, S., Chadha, R., Chiang, C.J.: An Intergrated Testbed for Virtual Ad Hoc Networks. In: 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities Internet, Washington D.C, pp. 1–10 (2009)

[12] Radkov, P., Yin, L., Goyal, P., Sarkar, P., Shenoy, P.: A Performance Comparison of NFS and iSCSI for IP-Networked Storage. In: 3rd USENIX Conference on File and Storage Technologies, San Francisco, California (2004)

[13] Geambasu, R., John, J.P.: Study of Virtual Machine Performance over Network File System. University of Washington Technical Report (2006)

[14] Lu, Y., Du, D.H.C.: Performance Study of iSCSI-Based Storage Subsystems. IEEE Communications Magazine, 76–82 (2003)

[15] Clark, T.: Storage virtualization: technologies for simplifying data storage and management. Addison Wesley (2005)

[16] VMWare: VMWare Virtual Machine File System: Technical Overview and Best Practices. VMWare Technical report, `http://www.vmware.com/pdf/vmfs-best-practices-wp.pdf` (retrived on November 30, 2010)