# Multi-Layered Virtual Machines for Security Updates in Grid Environments

Roland Schwarzkopf, Matthias Schmidt, Niels Fallenbeck, Bernd Freisleben

Department of Mathematics and Computer Science, University of Marburg
Hans-Meerwein-Str. 3, D-35032 Marburg, Germany
{rschwarzkopf,schmidtm,fallenbe,freisleb}@informatik.uni-marburg.de

***Abstract*—The use of user specific virtual machines (VMs) in Grid and Cloud computing reduces the administration overhead associated with manually installing required software for every user on every computational resource. However, a large number of user specific VMs increases the risk of security attacks. In particular, Cloud computing providers like Amazon suffer from these problems, since they offer different operating systems within VMs and delegate the security update problem for VMs to the users. In this paper, a solution that solves the problem by separating a VM into several layers is presented. The approach creates the possibility of installing security updates into a base layer centrally, affecting all VMs without affecting the users' own installed software stack by merging package databases. The proposal permits resource providers to keep a large number of VMs patched with the latest security fixes without bothering the users. Furthermore, the proposal avoids the overhead for transferring possible large VM images over the network between the nodes of a Grid or Cloud by allowing to hold locally cached VM images with a basic operating system installation while only the user-specific software stack stored in a separate layer needs to be transferred.**

## I. Introduction

Different users of a Grid or Cloud require different types of software installed on the computational resources to be able to execute their jobs. This is a challenge to the administrators of the participating sites, since every requested piece of software has to be installed and tested manually on all involved sites. Under certain circumstances, installation of individual software might not be possible at all, because conflicts with already installed software exist.

To solve this problem, the Xen Grid Engine (XGE) has been developed [1]. It uses user-specific Xen virtual machines (VMs) to provide a secure execution environment for jobs in a Grid. This is achieved by extending the Sun Grid Engine (SGE) [**?**], a cluster scheduler that is used as a backend by the Grid middleware. To simplify the process of creating a custom VM, the Image Creation Station (ICS) has been developed [1]. Using either a web interface or a service-oriented interface, a user can create a VM containing a base installation of a Linux operating system within a few minutes. After the creation, the user can remotely log in to the machine and install arbitrary software. When a job is scheduled to specific worker nodes in a computational cluster, the XGE distributes the corresponding VM image to the worker nodes, boots the VM and executes the job in this VM. Inside the VM, the user can access his or her home directory, mounted via the Network File System (NFS).

The use of NFS as the VM storage may also be an alternative to the VM image distribution method used by the XGE, especially at individual sites that provide high-performance Storage Area Networks (SAN) or equivalent technologies. However, for a large Grid with geographically distributed sites, NFS is not suitable. Usually, the inter-site connections provide less bandwidth than local connections, their usage is more expensive, and finally there might be security issues due to the unencrypted nature of NFS. Thus, Virtual Private Networks (VPNs) have to be used, causing a further degradation of performance. While this is unavoidable for the home directories, especially when files in the home directory are used for synchronization of compute jobs in VMs at different sites, NFS should not be used for hosting the VM images. Furthermore, network file systems are problematic if additional compute resources outside the computation center of an individual site (e.g. a pool of desktop computers) are included in the computation, because of firewalls or connections with smaller bandwidth. Finally, once transferred to a node, VM images can be cached locally, so they can be reused if a corresponding job is scheduled to this node again.

The remaining issues with the distribution of the VM images are bandwidth and transfer volume. While this is of less importance inside a data center or even between a data center and a desktop Grid within a single company, it becomes cumbersome when many images must be transfered between different computing centers.

The ICS provides a simple interface to VM creation, allowing even non-administrators to create VMs for their custom needs. These users typically do not have in-depth experience with system administration, especially with respect to security updates. It can be argued that security updates are not absolutely necessary for the worker nodes at Grid sites, because they are usually part of a cluster that is almost completely sealed from the outside by firewalls. However, sometimes security updates are published that have to be installed even in this scenario, e.g. the OpenSSH bug update for Debian GNU/Linux [2]. This bug reduces the possible number of generated SSH keys to 65535, allowing an attacker to gain root access usually in less than 20 minutes using precalculated keys [3]. Using the powerful machines at a Grid site may reduce the time even further, allowing attacks between the nodes of that Grid site.

A large number of users together with the possibility of having more than one VM per user increases the probability that some of the VMs will not be updated by their owners. This is not only a risk for those machines, but also for other VMs running. A single compromised VM can be used to attack other components at the same Grid site. This scenario not only affects a virtualized Grid installation, it also becomes a major concern of Cloud computing providers. Cloud computing uses virtualization technology to offer a non-shared use of computer resources with publicly accessible worker nodes on demand. For example, Amazon EC2 [**?**] offers VMs from their own organization, meaning that the majority of users operate on the same basic installation.

The solution proposed in this paper (1) improves the transfer of VM images at single Grid sites and between multiple Grid sites, and (2) provides administrators the option to centrally update a large number of VMs in case vital security updates are published. The basic idea of the approach is to separate the common and user specific parts of the VM images into distinct layers. Each of these layers hosts a complete filesystem, containing the base installation (base layer) or the software installed by the user (user layer), respectively. Using Copy-on-write (COW) filesystems like UnionFS [4], these layers can be merged into a single, virtual file system with the contents of both layers. Since the base layer is common to all VM images, it only needs to be transferred once to individual nodes or remote Grid sites, saving network resources. Only the (usually smaller) user layer needs to be transferred after a new VM is created or an existing VM is changed.

The separation of base installation and VM specific software allows an administrator to apply security updates to the base installation. This is a one-time task that automatically affects all VMs build upon this base installation. Obviously, this is not a strategy suitable for everyday updates, since the modification of the base installation without knowledge of the software installed on top of it may break dependencies, create conflicts and corrupt the software installed by the user. But in the case of well selected, necessary security updates as the one mentioned above that do not modify the functionality of the base installation except repairing the flaw, this is a feasible approach. After applying such an update, the package database of the VM is inconsistent, because only the database of the base installation is updated, which might not be visible in the VM images based on top of it, because of the UnionFS semantics. Thus, the databases of all VMs have to be merged with the one in the corresponding base installation.

The paper is organized as follows. Section II presents the proposed design. The implementation is discussed in Section III. Experimental results are presented in Section IV. In Section V, related work is discussed. Section VI concludes the paper and outlines areas for future research.

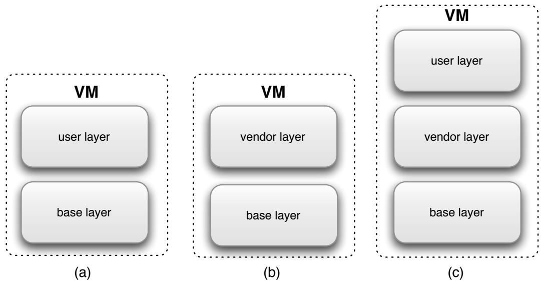


Fig. 1. Usage scenarios for a layered VM

## II. Design

In this section, the design of our proposed solution for the stated problems using a multi-layered root filesystem (MLRFS) in a VM and the process of updating individual layers, including potential problems and usage guidelines, are presented.

### A. *Multi-layered Root Filesystem*

A layered filesystem is a virtual filesystem built from more than one individual filesystem (layer) using a COW solution like UnionFS. The term multi-layered expresses the possibility to include three or more layers into that filesystem in a flexible way, to be usable in different scenarios. Finally, the term root filesystem expresses the use of the virtual filesystem itself as a root filesystem, in contrast to applying layers to individual folders.

There are different usage scenarios for layered VM images, as shown in Figure 1. The user may set up the user specific part of a VM completely on his or her own, leading to a two-layered image (a). Alternatively, a software vendor (or one of its sales partners) may provide a layer containing one of its products (b). Even in this case, the user may set up a custom layer on top of the vendor layer, containing extensions or other required tools (c). Nevertheless, other usage scenarios might be possible and should be supported.

Usually, a large number of similar jobs are submitted to a Grid, where each job represents a part of the problem to be solved. A simulation process is typically divided into numerous independent tasks that will be concurrently executed on many compute nodes. These jobs are executed in multiple instances of the VM, and depending on the scheduler used, they are most likely executed consecutively. Retransmitting the user layer again every time is contradicting to the whole idea of image distribution compared to the use of NFS. Thus, the user layer (as well as the base and possible vendor layers) should be cached locally on the individual nodes. To ensure that a cached layer is in a working state when it is used to form a root filesystem, it is best to make sure that it is not changed with respect to its original state. This can only be achieved by prohibiting write access to the layer during runtime.

*1) Building the MLRFS:* Considering the boot process of Linux, the MLRFS must be built before the *init* process is executed to start the system. This allows the user to make arbitrary changes to its VM, including changes to the service configuration or even the *init* process itself. Thus, building the root filesystem has to be done from inside the *initial ramdisk*, a minimal root filesystem that is responsible for mounting the real root filesystem. Inside the *initial ramdisk*, arbitrary scripts can be executed, as long as they are compatible with the restricted command set and shell.

The script must provide the following capabilities:

(1) Creating a root filesystem of an arbitrary number of layers
(2) Creating ramdisks to be used as layers
(3) Injecting files into the root filesystem
(4) Executing scripts (inside layers) before executing *init*

(1) ensures the flexibility needed to be usable in the different usage scenarios of layered VMs defined above. The use of ramdisks (2) is a possible solution to the use of read-only layers, which is explained in detail below. (3) provides a simple way to install site-specific configuration files into the root filesystem. In combination with (2), the files can be put in a dynamically created layer, together will all files that are changed during runtime of the VM, if the actual layers are mounted read-only. If configuration files do not provide enough possibilities to prepare a VM for execution at a specific Grid site, (4) allows the site administrator to execute arbitrary scripts inside the VM before the *init* process is executed. This is necessary to change existing files instead of completely overwriting them. For example, fixing the Debian OpenSSH bug requires to exchange all keys created with the unpatched version. This includes exchanging the public key of the ICS, installed in the `authorized_keys` file of the root account during VM creation that needs to be replaced without touching other public keys probably installed by the user.

*2) Read-only Layer Access:* To ensure reusability of layers cached at individual nodes, write access to these layers has to be prohibited during usage. Nevertheless, a running system needs write-access to the root filesystem. Thus, a single writeable layer must be part of the MLRFS. This can either be a ramdisk or a temporary layer stored locally on the node. The former approach is not suitable for Grid computing, where the users applications likely consume much memory for their applications. Obviously, the latter approach also has a drawback: an empty layer has to be created before the VM can be booted.

All files not residing in the user's home directory will be lost after the machine is shut down. To allow the user to analyze errors or check the execution of his or her jobs, the log files have to be saved to a persistent storage area during the machine is shutting down. This also applies to the use of temporary layers instead of ramdisks, because they are kept only during runtime and will eventually be reused for other VMs, after having been completely overwritten and reinitialized with a blank filesystem. This security precaution to prevent malicious users from trying to restore data from a temporary layer, is done in the background. Multiple temporary layers exist that are used successively to avoid idle time while the layer is cleared.

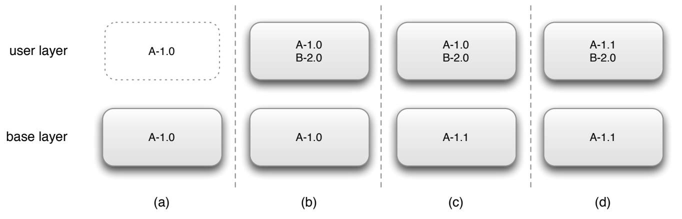


Fig. 2. Package databases in a dual-layered VM. A and B are packages with their version numbers.

*3) Update Layers and Merging:* When a layer is updated, it has to be transfered to all remote Grid sites using it. This involves marking all copies of that layer inside the caches of nodes as invalid, forcing the nodes to fetch the layer again. While the latter is usually a fast process, because the connection between the nodes and the layer storage of that the site normally has a high bandwidth, the former implies transferring layers over wide area networks with lower bandwidth.

Instead of directly updating a layer, the layer can be mounted read-only inside a MLRFS, together with a writable update layer on top. All changes occurring during the update are then stored inside the update layer, which is much smaller than the complete updated layer, because it only contains the changed files. Transferring the update layer to remote sites is thus a much faster process. At the remote site, the layer is updated using the contents of the update layer.

## B. Updating Virtual Machines

The MLRFS allows efficient updating of VMs, because the update only needs to be applied to a shared base layer centrally. Thereby all VMs based on that layer are automatically updated. As stated above, updating without knowledge of all installed software is problematic. Thus, (1) adequate care is required when updating a base layer and (2) VMs using the updated layer should be checked for compatibility, i.e. the absence of broken dependencies and conflicts.

Even when the update is compatible, the package database of the VM is very likely inconsistent. Figure 2 illustrates this problem. When the user installs software in his or her layer (b), UnionFS copies the package database from the base layer into the user layer before any changes are made to it. Thus, any changes in the package database of the base layer are not visible anymore (c). Merging the databases tends to provide a consistent package database in the user layer again (d).

A consistent package database is only needed when the VM is booted by the user on the ICS for maintenance reasons, i.e. for adding or updating software. On the other hand, when the machine is booted to execute a job, inconsistencies in the package database are irrelevant, because already installed software is used, but new software is not installed. In that case, a compatibility check is sufficient, whereas in the former case additional merging of the package database is required.

The merging of the package database is presented first, because the compatibility check can only be done on a consistent package database.

*1) Merging the Package Database:* In the merge process, the package databases of the old and the updated base layer are compared entry by entry to find updated, added or removed packages. The correct package database entry for each of those packages can be determined according to the corresponding flow chart. For every unmodified package, the entry in the user layer is kept. The remaining parts of the flow charts check whether the update can be applied successfully. For example, updated files may not be visible, even when the update is compatible in the sense stated above. This can either be caused by other versions of the files existing in the user layer, or because of so called whiteouts, the method used in UnionFS to delete files from lower, read-only layers.

Figure 3(a) shows the flow chart for packages updated in the base layer. The first step is the comparison whether the entry in the user layer package database is modified compared to the one in the old base layer. If the entries are equal, the files of that package likely exist only in the base layer and thus are visible. Unfortunately, there is a rare case in which the files of the package might exist in the user layer as well, even if the package database entries are equal: The user might have uninstalled the package and installed it again using the same version. The existence of the files from that package in the user layer has to be checked, because they would hide the updated ones. If the files exist in the user layer, the success of the update depends on the version comparison (see below). Otherwise, the update succeeded, if it is compatible. Anyway, the package database entry from the updated base layer is the correct one.

In case of a modified entry, the package has been either updated/downgraded or removed. In both cases, the updated files in the base layer are hidden, either by the ones in the user layer or by the corresponding whiteouts. The package database entry from the user layer is the correct one in both cases. In case the package was removed, the update is considered as failed and the chance of an incompatibility is high. Otherwise, the version comparison needs to be done.

In case the package is installed in both updated base and user layers, the package versions need to be compared. If the version in the base layer is less or equal, the update obviously succeeded, although it may be incompatible in the former case. On the other hand, if the version in the base layer is greater, the update is considered as failed and possibly is incompatible.

Figure 3(b) shows the flow chart for packages added in the base layer. It has to be checked if a corresponding entry exists in the user layer package database. If such an entry does not exist, the package was successfully added, although it may still be incompatible. The package database entry from the updated base layer is the correct one.

In case such an entry exists, the added package is either already installed in the user layer or has been installed and was removed. In the former case, the files added in the base layer are hidden by the ones in the user layer. Thus, the package database entry from the user layer is the correct one and the version comparison described above needs to be done. In the latter case, the files added to the base layer are visible. Thus, the package database entry from the updated base layer is the correct one and the package was successfully added, but may again be incompatible.

Figure 3(c) shows the flow chart for packages removed in the base layer. The first step is the comparison whether the entry in the user layer package database is modified compared to the one in the old base layer. If the entries are equal, the package was likely installed only in the base layer and thus can be removed. Except for possible incompatibilities, the removal succeeded and the base layer entry is correct. As for updated packages, it is possible that files of the package might exist in the user layer as well. In that case, the package would still be available. Thus, the removal failed, the user layer entry is correct and the update is possibly incompatible.

In case of a modified entry, the package has been either updated/downgraded or removed. The package database entry from the updated base layer is the correct one in both cases. If the package was removed, the removal in the base layer succeeded. Otherwise the removal is considered as failed and the probability of incompatibility is high.

*2) Checking Compatibility:* With a consistent package database, the compatibility check is a simple task. For each updated or added package, it must be checked that all dependencies are satisfied and conflicting packages are not installed. This is very likely the case, because these issues are already addressed during the update of the base layer. Nevertheless, the user might, for example, have removed a required package in the usage layer. Additionally, for updated packages it must be checked that none of the other packages conflicts with the specific version and that no other package depends exactly on the older version. Both cases are very rare, because the most conflicts with version constraints emerge with versions older than a specific version and the most dependencies require a version equal or newer than a specific version. If those constraints were satisfied for the old version, they are very likely also satisfied for the updated version. For added packages it must be checked that no other package conflicts with this package. For removed packages, it must be checked that no other package depends on the removed package. The latter two kinds of problems are more likely than the ones with updated packages.

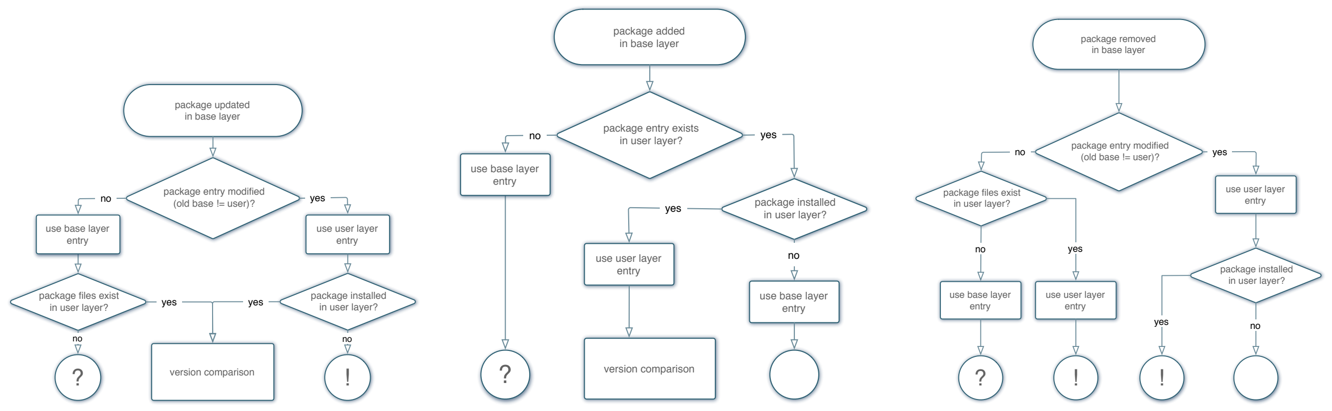
## III. Implementation

In this section, the implementation of the layered VMs and the proposed update mechanism that allows installation of security updates without endangering the integrity of the users personal software stack as well as the integrity of the base VM are presented.

### A. Multi-layered Root Filesystem

*Creating a root filesystem with an arbitrary number of layers.* Using boot parameters, a comma-separated list of devices and another single device can be set as read-only and writeable layers, respectively.

The devices are mounted in the specified order at mount-points created inside the *initial ramdisk* that serves as the root


package updated in base layer
package entry modified (old base != user)?
no
yes
use base layer entry
use user layer entry
package files exist in user layer?
package installed in user layer?
yes
yes
no
no
?
version comparison
!
package added in base layer
package entry exists in user layer?
no
yes
use base layer entry
package installed in user layer?
yes
no
use user layer entry
use base layer entry
?
version comparison
package removed in base layer
package entry modified (old base != user)?
no
yes
package files exist in user layer?
use user layer entry
no
yes
use base layer entry
use user layer entry
package installed in user layer?
yes
no
?
!
!

(a) Chart for updated packages (b) Chart for added packages (c) Chart for removed packages

Fig. 3. Flow charts of the package database merging process

filesystem until a chroot into the real root filesystem is done and the real *init* process is started. Thereafter, the MLRFS is built from all the layers using UnionFS and mounted at a specific mountpoint. Later on, the *init* process of the *initial ramdisk* does the chroot into the MLRFS.

The final steps are minimal, distribution dependent changes. If a / entry exists in `/etc/fstab`, it must be removed. During the normal startup, the root is mounted read-only first, checked for errors and remounted as writeable. This causes an error with the MLRFS, because it is already mounted as writeable. Additionally, the *init script* that executes the check of the root filesystem needs to be disabled.

*Creating ramdisks to be used as layers.* Using a special keyword instead of a device for the writeable layer or omitting the parameter completely uses a ramdisk as the writeable layer. Another parameter can be used to specify the size of the ramdisk. The ramdisk is implemented as *tmpfs*, the recommended ramdisk implementation of Linux, because it is very memory efficient.

Again, a distribution dependent change is necessary. When the system is shut down, at some point the swap is turned off. If the contents of the ramdisk require more memory than is available without swap, this will result in a kernel panic. Normally, all ramdisks are unmounted before the swap is turned off, but this obviously must fail for a ramdisk used as part of the MLRFS. Thus, a special *init script* that cleans up the ramdisk is created and added to the shutdown procedure.

*Injecting files into the root filesystem.* Using a parameter, a source can be specified that contains files to be inserted into the writeable layer of the MLRFS. Possible sources are devices and archives on a TFTP server. All files residing on the device or in the archive that is fetched after the first ethernet device is configured using DHCP are copied to the root of the writeable layer.

*Executing scripts (inside layers) before executing init.* Using a parameter, the layers desired to be searched for pre-*init* scripts can be specified, using their device name. The layers are scanned in the order they are specified, searching for scripts inside a special folder. Using a mechanism similar to *System V init*, the order of execution of the scripts is defined.

*Implementation issues.* When the chroot is done, the mountpoints of the indiviual layers are not visible anymore preventing access to the contents of individual layers. Unfortunately, this causes a problem during the shutdown. The individual layers cannot be unmounted, so they are remounted read-only, which obviously affects only the writeable layer. But when the mountpoints are not visible, remounting the writeable layer fails. Thus, the writeable layer is not cleanly unmounted, which may cause problems and requires a filesystem check to be done. To solve this, the mountpoints are moved inside the MLRFS, accepting that the individual layers are accessible.

## *B. Updating Virtual Machines*

The implementation of the merging and compatibility checking has been written in Python, based on the Debian Package Manager *dpkg*, which is also used by other distributions based on Debian. The package database used by *dpkg* consists of several parts located in `/var/lib/dpkg`, of which only three are considered for merging.

The `status` file is the main part of the database. It contains package information (e.g. name, version, dependencies, etc.) and its installation status. The the `info` directory contains file lists, checksums, lists of configuration files and installation/removal scripts for every package. Finally, the `diversions` file contains records about files from one package that have been replaced by files from another package. This allows to keep the changed files when the original package is updated.

The description of the merge process in the last section applies to the `status` file. The different records of each package are stored in terms of key-value assignments, one record per line. A blank line is used to separate two packages. For easy access to this file, a package has been written that simplifies the processing of this database. Classes exist for the package database as whole, providing a dictionary interface for easy package lookup, for individual packages, allowing easy checking of their state and accessing individual records, and even for individual records of a package, like dependencies or conflicts. The latter provide specialized functionality to simplify tasks like dependency checking or searching for conflicts.

Using the flow charts of the last section, the actual merging process is done using the classes described above. While the base and user layer package databases are located in the corresponding layers, a copy of the package database from the old base layer has to be used, because the layer does not exist anymore. Efficient checking for compatibility is more difficult. The easiest way is to do a full constraint check, i.e. check every package for unsatisfied dependencies and emerging conflicts. This can be a time consuming task for large package databases, especially because the comparison of versions is done using *dpkg*. The overhead of calling an external binary is accepted here to make sure the different version schemes used, even within a single distribution, are handled correctly. The full check also causes some problems, because the package database may contain some quirks like packages conflicting with themselves even in a plain base installation. Additionally, the user may, for example, force the installation of a package, even if this introduces a conflict. This would lead to an error message during the compatibility check although it might work. The efficient way described in the design section uses knowledge about the changes in the package database to reduce the number of expensive checks. Because it only does checks related to changes, it is usually able to handle even those problematic cases.

The `info` directory contains different files for each package that store the different types of information. This part of the package database must not be merged, because it always represents the actual packages visible to the system. Because of the UnionFS semantics, an older package installed in the user layer may hide a newer package in the base layer. In this case, the information in the `info` directory is correct, but the update in the base layer can not be applied. This is an error condition reported by the merging process.

Finally, the `diversions` file contains information about files replaced by different packages. For each replaced file, it stores the name of the package that replaced the file, the original name and the name of the backup file. While the replacement of files from another package is rare during a normal update, it might occur when new packages need to be installed within the scope of an update. The merging process is again driven by the semantics of UnionFS: Whenever conflicting records exist in the base and user layer versions of the `diversions` file, the records from the user layer are adopted.

## IV. Experimental Results

Adding COW-layers to VMs using UnionFS produces additional costs when performing I/O intensive tasks. We conducted a measurement to investigate this overhead. The used VMs have 128 MB RAM and a single assigned CPU core. The bonnie++ [**?**] benchmark is a well-known testing suite aimed to perform a number of filesystem related tests. It performs a series of tests on a file. For each test, it reports the number of kilobytes processed per elapsed second. We performed several tests: (1) the file is written *block* by block, (2) the file is written *character* by character and (3) the file is *rewritten*, requiring a read-, seek- and write-operation. Since no space allocation is done, and the I/O is well-localized, this should test the effectiveness of the filesystem cache and the speed of data transfer. A total of 100 tests were performed and the average of the results are shown in Figure 4. When writing the file block by block, the non-layered VM outperformed the layered VM (206277 KB/s vs. 198015 KB/s), thus the COW-layer introduces a slight performance reduction. The character test does not reveal any notable difference (48775 KB/s vs. 48319 KB/s), whereas in the rewrite test the layered VM had a significant higher throughput than the non-layered VM (67951 KB/s vs. 33472 KB/s). This is due to the COW-cache of the UnionFS file system. As a conclusion it can be stated that the introduction of the additional layers consumes some performance if files are written in large blocks. Once this step is performed, the performance benefits from the effective caching of the layered filesystem are evident. Due to the fact that most files of a regular job are written in the user's home directory that is natively accessible, the overhead only comes into play in certain, special circumstances.

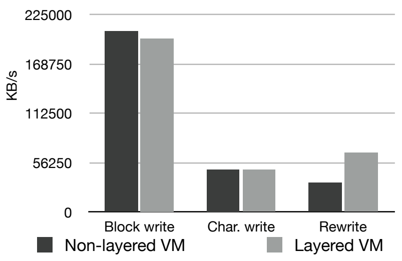


Fig. 4. Results of the bonnie++ benchmark with layered and unlayered VMs

Comparisons to similiar solutions have not been done for different reasons. Since the ability to update the base layer is an important feature of our proposed solution, only the solutions in [10], [11], [12] are comparable. Unfortunately, the former two solutions are not available for download, while the latter also uses UnionFS to create a layered filesystem, so the performance is expected to be identical.

Furthermore, we applied our solution to a real-world scenario. A user creates a VM and installs the package (s)he requires to run her or his jobs. If a job is executed, this VM needs to be transferred either to compute nodes of the Grid site where the VM was created or to a remote Grid site. Afterwards, a security update is applied to the VM, making it necessary to retransfer the VM.

Without the MLRFS, every time the user updates his or her VM and submits a job, the complete disk image of this machine has to be copied to the particular nodes (because a VM that was cached earlier is marked as invalid after the software update). When the MLRFS is used, only the user layer needs to be copied, which is significantly smaller. The base layer needs to be copied only once, because it is cached locally at the compute nodes. In case of an update, only the changes are copied and merged into the corresponding layer.

We conducted a measurement of the transfer time in both cases, comparing the VM with and without the MLRFS. The base installation consists of 162 packages using about 468 MB, and the user installs 14 additional packages using about 58 MB. The update includes 3 updated and 1 added package, using about 12 MB plus about 40 MB Debian Package Manager metadata (package lists, etc). Both VMs use a 4 GB sparse disk image containing an ext3 filesystem. The difference between the used space within the images and the sizes is caused by filesystem structures and the space allocated for files that have been deleted afterwards. Additionally, we measured the time needed to copy the changes of the update into the layer and to merge the package database including a compatibility check.

Since sparse files are used for the disk images, only the used parts of the image need to be transferred. For the transfer operation, `tar` is used over a SSH-encrypted data channel, because `tar` is able to handle sparse files efficiently and supports compression that speeds up the copy operation on slow networks. Using `scp` is not possible, because it does not support sparse files and would thus copy the complete files including unused parts.

| | Size (MB) | Transfer times | | |
|---|---|---|---|---|
| | | single site | multi site | |
| | | | uncompressed | gzipped |
| disk image | 691 | 40.59 secs | 660.83 secs | 460.12 secs |
| base layer (BL) | 666 | 39.12 secs | 636.92 secs | 443.47 secs |
| BL update | 72 | 15.06 secs | 106.23 secs | 100.11 secs |
| user layer | 67 | 14.45 secs | 101.51 secs | 91.58 secs |

TABLE I
TRANSFER TIMES OF VIRTUAL MACHINE DISK IMAGES AND FILESYSTEM LAYERS

Table I shows the measured time needed to transfer different images from one compute node to another without compressing the data during the copy process. We conducted 60 transfer operations to calculate a robust mean value. The used compute nodes are dual AMD Opteron nodes with 16 GB RAM each, interconnected with a GBit switched ethernet network. When VM images must be copied between Grid sites, the time needed for the copy operations increases dramatically. The table also shows the measurements of uncompressed and gzip-compressed data transfer between compute nodes on two different academic locations connected by the German Research Network (DFN).

Applying the update to the base layer took 4.05 seconds, merging the package database took 0.36 seconds. To check the scalability of the merge algorithm, we conducted a second measurement using a user layer containing 1231 packages instead of 176. The time needed to merge the package database grew to 1.01 seconds. All values are the means of 60 measurements.

Summing up, without the MLRFS the amount of data to be transferred for the VM including the update is about 1380 MB, taking about 81, 1330 or 925 secs (LAN, WAN, WAN compressed). Using our solution, the amount of data reduces to 140 MB, taking about 34, 212 or 196 secs when the base image is already cached or 805 MB and 73, 850 or 640 secs otherwise, although the latter case should be rare. This means that the use of the MLRFS saves up to 90% traffic and 60% – 85% time in the scenario.

## V. Related Work

There are several papers addressing storage virtualization to handle the problem of storing a large number of VM images efficiently. Typically, the use of snapshots (persistent views of a VM image at specific points in time) as the base for the creation of new VMs is proposed. VirtuaLinux [5] uses the Enterprise Volume Management System (EVMS) [6] as its storage. If a snapshot used as base of some VM images needs to be updated, all VM images based on it need to be recreated, making this approach unusable for centrally applied security updates. Additionally, to the best of our knowledge, EVMS provides no easy way to extract the differences between a snapshot and the current state of a VM image, thus an efficient transfer of VM images is not possible.

Parallax [7], [8] uses a custom mechanism for storing VM images and creating snapshots. Template images are used to build new VM images that share common blocks. The paper gives no details about the transfer of individual images, because the authors propose the use of a central SAN as storage. Updating the templates is not intended by the authors, although the block-oriented nature of their storage solution probably leads to the same problems as with VirtuaLinux.

To reach the goal of fast migration of VMs, Sapuntzakis et al. [9] propose a similar concept. VM images are built from a hierarchy of disks that are combined using block-oriented COW techniques at runtime. Transfer of individual disks is obviously possible, because it is a requirement for migration. Updating individual disks from the hierarchy is intended, but requires recreation of all disks based thereon. Again, this technique is not usable for centrally applied security updates.

A completely different approach is used in Ventana [10]. Instead of VM images as a virtual counterpart to physical discs, views of a virtual filesystem are used. A view is a combination of one or more branches that are trees of files and directories. Additionally, Ventana provides a version history for each file as well as Access Control Lists (ACLs) at the file or branch level. A component outside the VM called Host Manager is used to provide the view as NFS share, while the actual data is stored on external metadata and object servers and accessed using a specialized protocol. This solution allows the reuse of common parts of a VM image, but relies on fast networks to be usable. Applying security updates to the VMs is not addressed in the paper.

XenoServer [11] uses NFS to access the root filesystem in the VM, which is provided by another VM called Stacking COW server running at the same host. The filesystem consists of a local template and one or more VM specific layers called overlays. These overlays are stored remotely and accessed via the Andrew File System (AFS). No details are given on how the actual filesystem is built, nor on the overlays itself (filesystem images shared via AFS or AFS shares). While the idea is generally comparable to that of the MLRFS, the additional VM needed to provide the filesystem over NFS causes a performance degradation, because every I/O operation not only involves a context switch to the Virtual Machine Monitor (VMM), but also to another VM that may in turn start a new I/O operation to get the data via AFS. The authors mention another mode of operation of XenoServer, where no Stacking COW server is required, but the filesystem is built inside the VM. Unfortunately, no details are given about this approach, except that the overlay is fetched from some network server. Again, updating VMs is not addressed at all.

A proposal for VMs for distributed workstations that can be used as Condor nodes or virtual cluster has been made by Wolinski et al. [12]. Besides features like automatic network configuration, IP over P2P, etc. the paper also introduces a layered filesystem based on UnionFS. While there are some similarities to the MLRFS, their solution lacks some of the features required in our scenario, as described in Section II: (1) the flexibility to use an arbitrary number of layers – their solution seems to be restricted to three layers, (2) the use of ramdisks to keep the actual layers read only, (3) injecting (configuration) files into the root filesystem, (4) executing scripts before the actual *init* process starts. While the authors mention the necessity of security updates, they do not address the topic except stating that a layer can be exchanged without data-loss in upper layers. They do not mention the possible problems resulting from the exchange of a layer, as described in Section I.

## VI. Conclusions

In this paper, we have proposed a separation of VMs into a common part containing a base installation (base layer) and a VM specific part (user layer) that may be further divided into various layers. This separation allows an administrator to centrally apply security updates to the base layer that affect all VMs build upon this base layer. The inconsistencies inside the package database that arise from an updated base layer have been identified, and an algorithm that fixes those inconsistencies by merging the changes in the base layer into the package database has been presented.

Furthermore, the proposed solution solves an important problem in using VMs in Grid computing. However, the transfer of huge VM images between different Grid sites is cumbersome and may influence other computations running concurrently. Our approach only requires a one-time distribution of a potentially huge base layer. To use a new VM in the Grid, only a much smaller user layer needs to be distributed. This reduces the network load of VM distribution and shortens the time between creation and use of a VM. Efficient distribution of updates – again only the changes need to be transferred – amplifies this advantage.

There are several areas for future research: (1) As Amazon's EC2 spreads further, installation of security updates in every single VM will become cumbersome. By using the MLRFS, a central application of security updates is possible, providing a surplus value to customers. Currently, the MLRFS can not be used with Amazon EC2, because the creation of custom *initial ramdisks* is restricted to Amazon EC2 and selected vendors [13]. (2) UnionFS is known to have some performance problems with read-only branches, which is especially bad due to the wide use of read-only layers with the MLRFSs. These problems are solved by aufs (Another UnionFS), thus a performance comparison between both filesystems is necessary. (3) Besides the Debian Package Manager, the RPM Package Manager is widely used. The possibility of migrating the proposed solution to this package manager should be evaluated.

## References

[1] M. Smith, M. Schmidt, N. Fallenbeck, T. Doernemann, C. Schridde, and B. Freisleben, "Secure On-demand Grid Computing," in *Journal of Future Generation Computer Systems*. Elsevier, 2008.

[2] "Debian Security Advisory 1576-1 OpenSSH – Predictable Random Number Generator," http://www.debian.org/security/2008/dsa-1576, May 2008.

[3] "Debian OpenSSL Predictable PRNG Bruteforce SSH Exploit," http://www.milw0rm.com/exploits/5622, May 2008.

[4] D. P. Quigley, J. Sipek, C. P. Wright, and E. Zadok, "UnionFS: User- and Community-oriented Development of a Unification Filesystem," in *Proceedings of the 2006 Linux Symposium*, vol. 2, Ottawa, Canada, July 2006, pp. 349–362.

[5] M. Aldinucci, M. Torquati, M. Vanneschi, and P. Zuccato, "The virtualinux storage abstraction layer for efficient virtual clustering," in *PDP*. IEEE Computer Society, 2008, pp. 619–627. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/PDP.2008.86

[6] "Enterprise Volume Management System," http://evms.sourceforge.net/.

[7] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand, "Parallax: Managing storage for a million machines," *Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, 2005.

[8] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield, "Parallax: virtual disks for virtual machines," in *EuroSys*, ser. Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008, J. S. Sventek and S. Hand, Eds. ACM, 2008, pp. 41–54. [Online]. Available: http://doi.acm.org/10.1145/1352592.1352598

[9] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, "Optimizing the migration of virtual computers," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 377–390, 2002.

[10] B. Pfaff, T. Garfinkel, and M. Rosenblum, "Virtualization aware file systems: Getting beyond the limitations of virtual disks," in *NSDI*. USENIX, 2006. [Online]. Available: http://www.usenix.org/events/nsdi06/tech/pfaff.html

[11] E. Kotsovinos, T. Moreton, I. Pratt, R. Ross, K. Fraser, S. Hand, and T. Harris, "Global-scale service deployment in the xenoserver platform," in *1st Works. on Real, Large Distrib. Sys., WORLDS 04*. San Francisco, CA, 2004.

[12] D. I. Wolinsky, A. Agrawal, P. O. Boykin, J. R. Davis, A. Ganguly, V. Paramygin, Y. P. Sheng, and R. J. Figueiredo, "On the design of virtual machine sandboxes for distributed computing in wide-area overlays of virtual workstations," in *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2006, p. 8.

[13] "Feature Guide: Amazon EC2 User Selectable Kernels," http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1345.