Philipps Universität Marburg

Virtual Machine Lifecycle Management in Grid and Cloud Computing

Dissertation

zur Erlangung des Doktorgrades der Naturwissenschaften
(Dr. rer. nat.)

dem Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg
vorgelegt von

Diplom-Informatiker (FH)
Roland Schwarzkopf
geboren in Bad Salzungen

Marburg, im November 2015

Vom Fachbereich Mathematik und Informatik der Philipps-Universität Marburg (Hochschulkennziffer 1180) als Dissertation am 2. Dezember 2015 angenommen.

1. Gutachter: Prof. Dr. Bernd Freisleben, Philipps-Universität Marburg
2. Gutachter: Prof. Dr. Manfred Grauer, Universität Siegen

Tag der Einreichung: 12. November 2015.
Tag der mündlichen Prüfung: 8. Dezember 2015.

Abstract

Virtualization is the foundation for two important technologies: Virtualized Grid and Cloud Computing. Virtualized Grid Computing is an extension of the Grid Computing concept introduced to satisfy the security and isolation requirements of commercial Grid users. Applications are confined in virtual machines to isolate them from each other and the data they process from other users. Apart from these important requirements, Virtualized Grid Computing also solves other issues associated with Grid Computing, e.g., the problem of software deployment. Cloud Computing is another paradigm for using remote resources. This thesis focuses on the Infrastructure as a Service model that combines some of the ideas of (Virtualized) Grid Computing with a new kind of business model that features on-demand provisioning of raw computing resources (virtual machines) based on a pay-as-you-go pricing model, i.e., customers pay only for their actual usage.

The use of virtualization technology increases the utilization of physical hosts and simplifies systems management compared to physical machines, e.g., by allowing users to clone a virtual machine or to create a snapshot of a virtual machine as a backup of its state before it is modified. However, not all challenges regarding virtualization are solved yet, and the dynamic nature of both Virtualized Grid and Cloud Computing poses new requirements on the technology.

This thesis addresses various aspects of virtual machine usage in Virtualized Grid and Cloud Computing environments. First, the lifecycle of virtual machines in these environments is analyzed and corresponding models are developed. Then, several issues are identified and solutions for these issues are proposed. The key areas this thesis focuses on are the storage, deployment, and execution of virtual machines. Both storage and deployment are negatively affected by the traditional, self-contained image format used for storing virtual machines: large image files that store the contents of virtual disks. This format prevents an efficient deployment and wastes storage space. Furthermore, the security of virtual machines in these environments is a cross-cutting concern affecting all three areas. For example, deployment processes should consider information about the security of a virtual machine image and the execution environment should provide means to monitor virtual infrastructures effectively.

This thesis proposes the concept of image composition that combines multiple layers to a composite disk image. This facilitates sharing of common parts and reduces the deployment times of virtual machines as well as their storage requirements. The Marvin Image Compositor, an implementation of this concept, is presented. Furthermore, this thesis introduces the Marvin Image Store, a storage system for virtual machines that replaces traditional image files by a specialized storage system that separately stores the data and metadata contained in an image file. To improve the security of virtual environments, four different proposals are made: The Update Checker is a system that enables scanning virtual machines for outdated software irrespective of their state. The second proposal is an approach for centrally updating virtual machines that are built with the image composition technique, i.e., installing updates a single time and at the same time affecting multiple virtual machines. The Online Penetration Suite is a system that can automatically scan virtual machines for vulnerabilities. The last proposal in the security context is a monitoring concept that is based on monitoring every layer of a virtualized system and facilitates automatic responses to detected incidents. Finally, a virtual machine migration approach is presented that is able to efficiently migrate a virtual machine in the absence of a shared storage system.

The main contributions of this thesis are: an analysis of the virtual machine lifecycle in Virtualized Grid and Cloud Computing environments for different usage models, the introduction of the Marvin Image Compositor and the Marvin Image Store that optimize storage and deployment of virtual machines, as well as multiple solutions for improving the security of virtual machines and their management. Design and implementation details as well as experimental evaluations are presented for each of the proposals. A summary of the contributions and a discussion of areas for future work conclude this thesis.

Zusammenfassung

Virtualisierungstechnologie ist die Grundlage für zwei wichtige Konzepte: Virtualized Grid Computing und Cloud Computing. Ersteres ist eine Erweiterung des klassischen Grid Computing. Es hat zum Ziel, die Anforderungen kommerzieller Nutzer des Grid hinsichtlich der Isolation von gleichzeitig ausgeführten Batch-Jobs und der Sicherheit der zugehörigen Daten zu erfüllen. Dabei werden Anwendungen in virtuellen Maschinen ausgeführt, um sie voneinander zu isolieren und die von ihnen verarbeiteten Daten vor anderen Nutzern zu schützen. Darüber hinaus löst Virtualized Grid Computing das Problem der Softwarebereitstellung, eines der bestehenden Probleme des klassischen Grid Computing. Cloud Computing ist ein weiteres Konzept zur Verwendung von entfernten Ressourcen. Der Fokus dieser Dissertation bezüglich Cloud Computing liegt auf dem „Infrastructure as a Service Modell", das Ideen des (Virtualized) Grid Computing mit einem neuartigen Geschäftsmodell kombiniert. Dieses besteht aus der Bereitstellung von virtuellen Maschinen auf Abruf und aus einem Tarifmodell, bei dem lediglich die tatsächliche Nutzung berechnet wird.

Der Einsatz von Virtualisierungstechnologie erhöht die Auslastung der verwendeten (physischen) Rechnersysteme und vereinfacht deren Administration. So ist es beispielsweise möglich, eine virtuelle Maschine zu klonen oder einen Snapshot einer virtuellen Maschine zu erstellen, um zu einem definierten Zustand zurückkehren zu können. Jedoch sind noch nicht alle Probleme im Zusammenhang mit der Virtualisierungstechnologie gelöst. Insbesondere entstehen durch den Einsatz in den sehr dynamischen Umgebungen des Virtualized Grid Computing und des Cloud Computing neue Herausforderungen für die Virtualisierungstechnologie.

Diese Dissertation befasst sich mit verschiedenen Aspekten des Einsatzes von Virtualisierungstechnologie in Virtualized Grid und Cloud Computing Umgebungen. Zunächst wird der Lebenszyklus von virtuellen Maschinen in diesen Umgebungen untersucht, und es werden Modelle dieses Lebenszyklus entwickelt. Anhand der entwickelten Modelle werden Probleme identifiziert und Lösungen für diese Probleme entwickelt. Der Fokus liegt dabei auf den Bereichen Speicherung, Bereitstellung und Ausführung von virtuellen Maschinen. Virtuelle Maschinen werden üblicherweise in so genannten Disk Images, also Abbildern von virtuellen Festplatten, gespeichert. Dieses Format hat nicht nur Einfluss auf die Speicherung von größeren Mengen virtueller Maschinen, sondern auch auf deren Bereitstellung. In den untersuchten Umgebungen hat es zwei konkrete Nachteile: es verschwendet Speicherplatz und es verhindert eine effiziente Bereitstellung von virtuellen Maschinen. Maßnahmen zur Steigerung der Sicherheit von virtuellen Maschinen haben auf alle drei genannten Bereiche Einfluss. Beispielsweise sollte vor der Bereitstellung einer virtuellen

Maschine geprüft werden, ob die darin installierte Software noch aktuell ist. Weiterhin sollte die Ausführungsumgebung Möglichkeiten bereitstellen, um die virtuelle Infrastruktur wirksam zu überwachen.

Die erste in dieser Dissertation vorgestellte Lösung ist das Konzept der *Image Composition*. Es beschreibt die Komposition eines kombinierten Disk Images aus mehreren Schichten. Dadurch können Teile der einzelnen Schichten, die von mehreren virtuellen Maschinen verwendet werden, zwischen diesen geteilt und somit der Speicherbedarf für die Gesamtheit der virtuellen Maschinen reduziert werden. Der *Marvin Image Compositor* ist die Umsetzung dieses Konzepts. Die zweite Lösung ist der *Marvin Image Store*, ein Speichersystem für virtuelle Maschinen, das nicht auf den traditionell genutzten Disk Images basiert, sondern die darin enthaltenen Daten und Metadaten auf eine effiziente Weise getrennt voneinander speichert. Weiterhin werden vier Lösungen vorgestellt, die die Sicherheit von virtuellen Maschine verbessern können: Der *Update Checker* ist eine Lösung, die es ermöglicht, veraltete Software in virtuellen Maschinen zu identifizieren. Dabei spielt es keine Rolle, ob die jeweilige virtuelle Maschine gerade ausgeführt wird oder nicht. Die zweite Sicherheitslösung ermöglicht es, mehrere virtuelle Maschinen, die auf dem Konzept der Image Composition basieren, zentral zu aktualisieren. Das bedeutet, dass die einmalige Installation einer neuen Softwareversion ausreichend ist, um mehrere virtuelle Maschinen auf den neuesten Stand zu bringen. Die dritte Sicherheitslösung namens *Online Penetration Suite* ermöglicht es, virtuelle Maschinen automatisiert nach Schwachstellen zu durchsuchen. Die Überwachung der virtuellen Infrastruktur auf allen Ebenen ist der Zweck der vierten Sicherheitslösung. Zusätzlich zur Überwachung ermöglicht diese Lösung auch eine automatische Reaktion auf sicherheitsrelevante Ereignisse. Schließlich wird ein Verfahren zur Migration von virtuellen Maschinen vorgestellt, welches auch ohne ein zentrales Speichersystem eine effiziente Migration ermöglicht.

Eines der zentralen Ergebnisse dieser Dissertation ist eine Analyse des Lebenszyklus von virtuellen Maschinen in Virtualized Grid und Cloud Computing Umgebungen für verschiedene Anwendungsszenarien. Weiterhin zählen der Marvin Image Compositor und der Marvin Image Store zu den zentralen Ergebnissen, zwei Lösungen zur Optimierung der Speicherung und Bereitstellung von virtuellen Maschinen. Zusätzlich werden verschiedene Lösungen zur Verbesserung der Sicherheit und des Managements von virtuellen Maschinen vorgestellt. Alle Lösungen werden detailliert mit Design, Implementierung und Evaluation beschrieben. Eine Zusammenfassung und ein Ausblick auf künftige Forschung schließen die Arbeit ab.

Erklärung

Ich versichere, daß ich meine Dissertation

Virtual Machine Lifecycle Management in Grid and Cloud Computing

selbständig, ohne unerlaubte Hilfe angefertigt und mich dabei keiner anderen als der von mir ausdrücklich bezeichneten Quellen und Hilfen bedient habe. Die Dissertation wurde in der jetzigen oder einer ähnlichen Form noch bei keiner anderen Hochschule eingereicht und hat noch keinen sonstigen Prüfungszwecken gedient.

Marburg, den 11.11.2015 Roland Schwarzkopf

This page is intentionally left blank.

Acknowledgments

First of all, I would like to thank Prof. Dr. Bernd Freisleben for supervising me over the course of my dissertation, for his assistance and the valuable discussions that helped me to advance this thesis.

I would also like to thank Prof. Dr. Manfred Grauer at the University of Siegen for kindly taking the time to act as a reviewer of my thesis and for his support during the three years I was part of his working group.

Furthermore, I would like to thank Prof. Dr. Helmut Dohmann at the Fulda University of Applied Sciences for encouraging me to pursue a dissertation and supporting me in the initial phase of the thesis.

During the work on this thesis, I was financially supported by the German Ministry of Research and Education (BMBF) and the Hessen State Ministry of Higher Education, Research and the Arts (HMWK).

I would like to thank my colleagues and students past and present at the Distributed Systems Group who were invaluable in the realization of the projects in this thesis (in alphabetical order): Lars Baumgärtner, Kay Dörnemann, Dr. Tim Dörnemann, Sascha Fahl, Dr. Niels Fallenbeck, Pablo Graubner, Katharina Haselhorst, Prof. Dr. Steffen Heinzl, Dr. Ernst Juhnke, Matthias Leinweber, Simon Martin, Dr. Markus Mathes, Mathias Rüdiger, Dr. Matthias Schmidt, Prof. Dr. Matthew Smith, and Christian Strack. I'm especially thankful to my long-term office mates Niels, Matthias, Kay, and Pablo for providing such a pleasant working environment. Thanks to you all for many interesting and helpful conversations, lots of fun and for a helping hand when time was running out occasionally. Additionally, I would like to thank Mechthild Kessler for being the "good soul of the working group" and taking care of all administrative tasks.

Moreover, I would like to thank Hans-Michael Mahr who has decisively influenced my decision to study computer science with his enthusiasm for this subject that he showed in every lesson.

I would also like to thank my parents for supporting me throughout my studies and always encouraging me to pursue my goals.

Finally, I would like to thank Sabrina for her support and patience during the work on this thesis.

This page is intentionally left blank.

Contents

"The last thing one discovers in composing a work is what to put first."
T. S. Eliot (1888–1965)

1 Introduction

More than one and a half decades ago, Foster and Kesselman published a book that described the fundamental concepts of Grid Computing [54]. In the following years, Grid Computing became an important concept in academia, because it enabled researchers of institutions without sufficient local resources to use the resources of other institutions for their research, e.g., compute intensive simulations. However, in Grid Computing resources are not limited to plain compute power, but also include storage space for data.

The popularity of the Grid among researchers soon aroused commercial interest. Commercial users in the Grid, however, had different requirements compared to researchers. The applications, data, and results of academic research are typically not confidential, mostly because academic research is depending on public funding and thus committed to publishing results. On the other hand, the applications, data, and results of commercial users are mostly confidential. The security mechanisms available in the Grid were not sufficient for the industry to adapt Grid Computing.

The need for improved security in Grid Computing became visible in three research projects with commercial partners. All three projects were part of the D-Grid Initiative [31] and funded by the Federal Ministry for Education and Research (Bundesministerium für Bildung und Forschung, BMBF) [20]. The *Business to the Grid* (Biz2Grid) project [18] aimed to fit the Grid into the organizational frameworks of commercial IT infrastructures and to develop billing and pricing models. However, in the course of the project the security aspect became more important for the partner from the automotive industry. No only was the data processed in the Grid considered sensitive, but also the information about the application used to process the data, because it could leak information about the progress of product developers to competitors. Strong security mechanisms to protect the data and strong isolation between users to prevent leaking of information were identified as critical requirements for the use of the Grid in the automotive industry. The *Financial Business Grid* (FinGrid) project [48] aimed to port selected business processes and services to the Grid as a prototype of novel Grid-based financial applications that have previously been impossible to implement because of restricted computation power. Because these applications process financial data of customers that is very sensitive, strong security mechanisms were one of the most important requirements in this project. The *Plasma Technology Grid* (PT-Grid) project [121] aimed to provide Grid-based modeling and simulation environments for plasma technology applications using software provided by the partners. Again, the data processed is sensible and requires strong security mechanisms to protect the intellectual property of customers.

The solution for the security and isolation requirements that emerged in these projects was the use of virtualization: the concept of *Virtualized Grid Computing*. It is an extension of the Grid Computing concept that confines applications in virtual machines to isolate them from each other and protect the data they process from other users.

Two other research projects that were not part of the D-Grid initiative strengthened the need for virtualization. The *Tools for Intelligent System Management of Very Large Computing Systems* (TIMaCS) project [155] aimed to reduce the complexity of administrating computing systems by introducing a management system that automates many tasks. Virtualization was used in this project to partition large systems, to facilitate migration of running applications in case of errors or for maintenance operations, and to enable temporary suspension of individual applications to clear space for massive-parallel jobs. The same goals were also pursued in the *Management virtueller Maschinen in Linux High Performance Clustern* project [139]. The former project was funded by the BMBF, whereas the latter was funded by the Hessen State Ministry of Higher Education, Research and the Arts (Hessisches Ministerium für Wissenschaft und Kunst, HMWK) [67].

A few years after the introduction of Grid Computing a new computing paradigm emerged: *Cloud Computing*. In this thesis, the term Cloud Computing is used as a synonym for the Infrastructure as a Service (IaaS) model. It combined some of the ideas of Grid Computing with virtualization and a new kind of business model that features on-demand provisioning of computing resources with a pay-as-you-go pricing model, i.e., customers pay only for their actual usage. These two properties of Cloud Computing were critical for its success with commercial users, because it solved many of the problems they had with the Grid.

Virtualization technology obviously plays an important role in those developments. The success of Cloud Computing shows that virtualization has many benefits both for customers and providers. However, not all problems resulting from the use of virtualization are solved yet and even some new problems are caused by its usage. An example of such a problem is the monitoring of virtualized infrastructures that differ from traditional computing environments by the existence of an additional layer (the hypervisor) and dynamics introduced by starting, stopping, or migrating virtual machines. The Mastering Security Anomalies in Virtualized Computing Environments via Complex Event Processing (ACCEPT) project [2] (funded by the BMBF) aimed to solve this problem by using a large number of sensors at every layer, Complex

Event Processing technology to correlate the sensor data, and a flexible framework for responding to anomalies.

This thesis addresses various aspects of virtual machine usage in Virtualized Grid and Cloud Computing environments. First of all, the lifecycle of virtual machines in these environments is analyzed. Based on this analysis, several issues are identified and solutions for these issues are proposed. The areas this thesis focuses on are the storage, deployment, and execution of virtual machines.

1.1 Contributions of this Thesis

The research contributions of this thesis are:

- The concept of Lifecycle Management is proposed in the context of virtual machines, by identifying two *usage models* and developing a corresponding *lifecycle* for each model. The model that applies to Virtualized Grid and Cloud Computing environments is examined in detail and its implications for the handling of virtual machines are discussed.

- Virtual machines are traditionally stored as self-contained image files that contain both the operating system and applications. Especially the operating system is very likely shared by multiple virtual machines. This thesis introduces the *concept of image composition* that combines multiple layers to a composite disk image. This allows sharing of common parts and reduces the deployment times of virtual machines as well as their storage requirements. Furthermore, this thesis introduces the *Marvin Image Compositor* as an implementation of this concept and evaluates its efficiency and performance.

- The traditional format used to store virtual machines are image files. As already stated, image files fail to utilize similarities between virtual machines. This is especially important if the version history of a virtual machine should be preserved by keeping its older versions in addition to the current one. This thesis introduces a *proposal for storing virtual machines* that is based on the idea of data and metadata extraction and separate storage of these two kinds of information in specialized storage systems. This reduces the storage requirements further than image composition and provides novel management, maintenance and analysis operations. Furthermore, this thesis introduces the *Marvin Image Store* as an implementation of this concept and evaluates its efficiency and performance.

- The use of virtual machines boosts the number of systems that need continuous maintenance to keep them secure. Unfortunately, not all virtual machines are running continuously, so traditional maintenance approaches cannot be used in every situation. This thesis introduces two proposals for keeping virtual machines up-to-date. The first proposal deals with *scanning for outdated software in dormant virtual machines*, whereas the second proposal deals with *centrally updating multiple virtual machines* using image composition. Furthermore, this thesis further introduces the *Update Checker* and a *set of tools* to support the centralized update process as implementations of these proposals.

- Even if a virtual machine contains the latest versions of the operating system and applications, misconfigurations or insecure services can still threaten its security. In this thesis, the *Online Penetration Suite* – a method for automated scanning of virtual machines using off-the-shelf vulnerability scanners – is presented and its performance is evaluated.
- Virtualization adds another layer to the software stack and traditional monitoring tools just ignore this layer – it is transparent to them. This thesis proposes a *novel concept for security monitoring* in Virtualized Grid and Cloud Computing environments that monitors all layers of the software stack and is able to respond to incidents automatically.
- Most hypervisors support migration of virtual machines only if their virtual disks are available to both the source and destination host of a migration. This typically requires a shared storage system. This thesis presents a *proposal for migrating virtual machines* in absence of such a shared storage system. Furthermore, it describes its implementation as part of a virtual machine management system and evaluates its performance.

The following papers were published during the course of the work on this thesis:

1. Roland Schwarzkopf, Matthias Schmidt, Christian Strack, Simon Martin, Bernd Freisleben. Increasing Virtual Machine Security in Cloud Environments. In *Journal of Cloud Computing: Advances, Systems and Applications*, 1(1), Springer, 2012
2. Roland Schwarzkopf, Matthias Schmidt, Mathias Rüdiger, Bernd Freisleben. Efficient Storage of Virtual Machine Images. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing (ScienceCloud '12)*, pp. 51–60, ACM, 2012
3. Lars Baumgärtner, Pablo Graubner, Matthias Leinweber, Roland Schwarzkopf, Matthias Schmidt, Bernhard Seeger, Bernd Freisleben. Mastering Security Anomalies in Virtualized Computing Environments via Complex Event Processing. In *Proceedings of the The 4th International Conference on Information, Process, and Knowledge Management (eKNOW 2012)*, pp. 760–81, XPS, 2012
4. Roland Schwarzkopf, Matthias Schmidt, Christian Strack, Bernd Freisleben. Checking Running and Dormant Virtual Machines for the Necessity of Security Updates in Cloud Environments. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 239–246, IEEE Press, 2011
5. Matthias Schmidt, Sascha Fahl, Roland Schwarzkopf, Bernd Freisleben. TrustBox: A Security Architecture for Preventing Data Breaches. In *Proceedings of the 19th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP '11)*, pp. 635–639, IEEE Computer Society, 2011
6. Katharina Haselhorst, Matthias Schmidt, Roland Schwarzkopf, Niels Fallenbeck, Bernd Freisleben. Efficient Storage Synchronization for Live Migration in Cloud

Infrastructures. In *Proceedings of the 19th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP '11)*, pp. 511–518, IEEE Computer Society, 2011

7. Niels Fallenbeck, Matthias Schmidt, Roland Schwarzkopf, Bernd Freisleben. Inter-Site Virtual Machine Image Transfer in Grids and Clouds. In *Proceedings of the 2nd International ICST Conference on Cloud Computing (CloudComp 2010)*, Springer LNICST, 2010

8. Eugen Volk, Jochen Buchholz, Stefan Wesner, Daniela Koudela, Matthias Schmidt, Niels Fallenbeck, Roland Schwarzkopf, Bernd Freisleben, Götz Isenmann, Jürgen Schwitalla, Marc Lohrer, Erich Focht, Andreas Jeutter. Towards Intelligent Management of Very Large Computing Systems. In *Proceedings of Competence in High Performance Computing (CiHPC)*, pp. 191–204, Springer, 2010

9. Ernst Juhnke, Tim Dörnemann, Roland Schwarzkopf, Bernd Freisleben. Security, Fault Tolerance and Modeling of Grid Workflows in BPEL4Grid. In *Proceedings of Software Engineering 2010, Grid Workflow Workshop (GWW 2010), Lecture Notes in Informatics (LNI), Vol. P-160*, pp. 193–200, Gesellschaft für Informatik (GI), 2010

10. Markus Mathes, Christoph Stoidner, Roland Schwarzkopf, Steffen Heinzl, Tim Dörnemann, Helmut Dohmann, Bernd Freisleben. Time-Constrained Services: A Framework for using Real-Time Web Services in Industrial Automation. In *Service Oriented Computing and Applications*, 3(4), pp. 239–262, Springer, 2009

11. Roland Schwarzkopf, Matthias Schmidt, Niels Fallenbeck, Bernd Freisleben. Multi-Layered Virtual Machines for Security Updates in Grid Environments. In *Proceedings of 35th Euromicro Conference on Internet Technologies, Quality of Service and Applications (ITQSA)*, pp. 563–570, IEEE Press, 2009

12. Markus Mathes, Roland Schwarzkopf, Steffen Heinzl, Tim Dörnemann, Bernd Freisleben. Composition of Time-Constrained BPEL4WS Workflows using the TiCS Modeler. In *Proceedings of the 13th IFAC Symposium on Information Control Problems in Manufacturing (INCOM)*, pp. 892–897, Elsevier, 2009

13. Tim Dörnemann, Markus Mathes, Roland Schwarzkopf, Ernst Juhnke, Bernd Freisleben. DAVO: A Domain-Adaptable, Visual BPEL4WS Orchestrator. In *Proceedings of the 23rd IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pp. 121–128, IEEE Computer Society Press, 2009, (*Highly Commended Paper Award*)

14. Markus Mathes, Steffen Heinzl, Roland Schwarzkopf, Bernd Freisleben. F&L-Grid: Eine generische Backup und Recovery Infrastruktur für das D-Grid. In *Tagungsband des 2. DFN-Forum Kommunikationstechnologien*, pp. 55–68, Gesellschaft für Informatik (GI), 2009

15. Matthias Schmidt, Niels Fallenbeck, Kay Dörnemann, Roland Schwarzkopf, Tobias Pontz, Manfred Grauer, Bernd Freisleben. Aufbau einer virtualisierten Cluster-Umgebung. In *Grid Computing in der Finanzindustrie*, Books on Demand,

Norderstedt, pp. 119–131, Oliver Hinz, Roman Beck, Bernd Skiera, Wolfgang König, 2009

16. Markus Mathes, Roland Schwarzkopf, Steffen Heinzl, Tim Dörnemann, Bernd Freisleben. Orchestration of Time-Constrained BPEL4WS Workflows. In *Proceedings of the 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–4, IEEE Computer Society Press, 2008

17. Roland Schwarzkopf, Markus Mathes, Steffen Heinzl, Bernd Freisleben, Helmut Dohmann. Java RMI versus .NET Remoting - Architectural Comparison and Performance Evaluation. In *Proceedings of the 7th International Conference on Networking (ICN)*, pp. 398–407, IEEE Computer Society Press, 2008

1.2 Organization of this Thesis

The rest of the thesis is organized as follows:

In Chapter 2, an overview of topics that lay out the foundations for this thesis is given. This includes Grid, Virtualized Grid, and Cloud Computing as well as virtualization.

In Chapter 3, the idea of Lifecycle Management is considered in the context of virtual machines. Two different usage models are identified and a corresponding virtual machine lifecycle is developed for each model. Finally, the implications of these models for the handling of virtual machines are discussed.

In Chapter 4, the concept of image composition is presented as a way to improve the deployment times of virtual machines. The design and implementation of this concept are discussed and the implementation is evaluated.

In Chapter 5, a proposal for storing a large number of virtual machine images including their version history is presented that delivers efficiency both in terms of storage requirements and access time. The design and implementation of this proposal are discussed and the implementation is evaluated.

In Chapter 6, four proposals are made to improve the security of virtual machines in Virtualized Grid and Cloud Computing environments. Two of the proposals are concepts for dealing with software maintenance, i.e., keeping the software installed in virtual machines up-to-date, the third proposal is a method for detecting vulnerabilities in virtual machines, and the fourth proposal is a novel concept for monitoring virtual machines during runtime. The design and implementation of these proposals are discussed and their implementation is evaluated.

In Chapter 7, a novel method for storage migration as part of virtual machine migration is presented. The design and implementation of this method are discussed and the implementation is evaluated.

Finally, Chapter 8 concludes this thesis and discusses future work.

"People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones."

Donald Knuth (1938–)

2 Fundamentals

2.1 Introduction

This chapter contains an introduction to the both environments the solutions proposed in this thesis are designed for – Virtualized Grid and Cloud Computing – and the fundamental technology these environments and the solutions are based on – Virtualization.

The first section gives a brief overview of Grid Computing, covering the basic concept, the security infrastructure and the fundamental concepts of job and data management. In the second section, two approaches for Virtualized Grid Computing, a combination of Grid Computing with Virtualization technology, are presented. Afterwards, the concept of Cloud Computing and the different service and deployment models are presented.

The last section of this chapter deals with Virtualization, which is the base of both Virtualized Grid Computing and Cloud Computing. This sections covers both the fundamentals of Virtualization technology and the different approaches for virtualization on the x86 and x86-64 Architectures.

2.2 Grid Computing

Grid Computing has been introduced to simplify the use of remote resources, e.g., computational resources or storage. The name has been chosen to express one of the aims of Grid Computing: the access of remote resources should be as easy as using energy from the power grid. The term Grid Computing was widely used for different technologies, some of which do not qualify as Grids according to the definition of

Foster, one of the pioneers of Grid Computing. He summarized earlier definitions in a three point checklist [51]:

> “ I suggest that the essence of the definitions above can be captured in a simple checklist, according to which a grid is a system that:
>
> 1) . . . coordinates resources that are not subject to centralized control . . . (A grid integrates and coordinates resources and users that live within different control domains – for example, the user’s desktop vs. central computing; different administrative units of the same company; or different companies; and addresses the issues of security, policy, payment, membership, and so forth that arise in these settings. Otherwise, we are dealing with a local management system.)
>
> 2) . . . using standard, open, general-purpose protocols and interfaces . . . (A grid is built from multi-purpose protocols and interfaces that address such fundamental issues as authentication, authorization, resource discovery, and resource access. As I discuss further below, it is important that these protocols and interfaces be standard and open. Otherwise, we are dealing with an application- specific system.)
>
> 3) . . . to deliver nontrivial qualities of service. (A grid allows its constituent resources to be used in a coordinated fashion to deliver various qualities of service, relating for example to response time, throughput, availability, and security, and/or co-allocation of multiple resource types to meet complex user demands, so that the utility of the combined system is significantly greater than that of the sum of its parts.)
>
> *Foster [51], pp. 2–3* ”

Distributed computing paradigms existing at the time Grid Computing was introduced did not meet all of the criteria mentioned in this checklist. For example, many of the solutions available at that time were restricted to resources within a single administrative domain. Grid computing got rid of this restriction. A number of protocols and tools have been developed by researchers to implement Grid systems according to this definition. The results of these efforts are Grid middlewares such as the *Globus Toolkit* [52], *gLite* [83], or *Unicore* [151].

These middlewares at least provide support for job and data management and implement a security infrastructure. Job management deals with the submission and management of compute jobs at Grid site, i.e., a computing system that is integrated into the Grid, whereas data management deals with moving the data required by a job to the target Grid site. Both of these tasks rely on a coherent authentication and authorization system to be in place at any Grid site. A short overview of these three pillars of Grid middlewares is given in the next two sections.

The fundamental concept of Grid Computing is coordinated resource sharing with the goal of solving problems in dynamic, *virtual organizations*. A group of individuals or institutions consisting of resource providers and consumers are called virtual organization. The sharing process is controlled, i.e., rules exists stating which resources are shared with whom under what conditions. In this context, resources refer to compute resources, software, data and other resources [55].

2.2.1 Grid Security Infrastructure

The Grid Security Infrastructure (GSI), formerly known as Globus Security Infrastructure, solves the fundamental needs for authentication, authorization and confidentiality in the Grid. A brief description of how these needs where satisfied is given below.

Authentication — The authentication in the Grid is based solely on X.509 certificates and thus each user of the Grid has to possess a valid certificate that is signed by the authorized certificate authority. These certificates – more precisely, the corresponding private key – are password protected to prevent their abuse by unauthorized users. However, this protection measure prevents the delegation of a user's identity that is often required in Grid Computing, e.g., a job running on a remote Grid site might need to transfer files on behalf of the user. The user cannot authenticate himself interactively in such a case, because his job might be scheduled at an arbitrary time.
This problem is solved in the GSI using so-called *proxy certificates*, i.e., temporary certificates that can be used to authenticate as a user. A proxy certificate is signed using the user's long-term private key, i.e., the private key belonging to the user's certificate. Thereby, it proves that a specifically generated key pair belongs to the user. The private key of this key pair, the proxy certificate, and the actual certificate of the user – but not his long-term private key – are passed to any identity that needs to impersonate as the user. The lifetime of this proxy certificate is very short, typically a few hours, to prevent abuse in case it falls into the wrong hands.

Authorization — Authorization of users is implemented using different technologies. The simplest one is the *Gridmap* file, a plain text database that maps the *distinguished name* contained in a user's certificate to a local user account for a single Grid site. The authorization is then delegated to the operating system. Additionally, there are more advanced techniques for authorization, e.g., the Community Authorization Service (CAS) or the Virtual Organization Membership Service (VOMS) [25].

Confidentiality — Confidentiality is provided using Transport Layer Security (TLS) for the entire communication between middleware components and different Grid sites. The existing certificate infrastructure used for authentication is also used for the encryption of network communication.

2.2.2 Grid Job and Data Management

The data management facility of a Grid middleware deals with moving data to and from remote Grid sites to facilitate the execution of jobs at these sites. Different techniques for data management are provided by the different middlewares. For example, the

Globus Toolkit provides both *GridFTP*, an extension of the *File Transfer Protocol* (FTP) that is integrated with the Grid Security Infrastructure to provide authentication and confidentiality, and the *Reliable File Transfer* (RFT) service, a service that enables reliable background transfers based on GridFTP [93].

Job management deals with the creation, management and deletion of jobs at remote Grid sites. This bridges the gap between the Grid middleware with its Grid-wide view and local batch schedulers that are responsible for managing the unattended execution of jobs at individual Grid sites (batch processing). The job management facility of a Grid middleware is not a scheduler itself, but an interface for different local batch schedulers. Typically, it is tightly coupled with the data management facilities of the middleware to automatically handle the transfer of job input data and results to and from the remote Grid site before and after the execution of the job, respectively.

2.3 Virtualized Grid Computing

While Grid Computing solves many of the issues that existed in older distributed computing paradigms, it leaves a few open problems. Two of the major problems of Grid Computing are security and isolation. Traditionally, the security of the grid is based on a trust relationship. The users have to trust the resource provider not to provide malicious resources and the provider has to trust the users not to perform malicious actions [46]. Furthermore, the operating system's security mechanisms are the only safeguards for the users' data against a malicious users.

Two other problems are related to provisioning the required execution environment (applications and libraries) and utilization of physical resources. Different virtual organizations often have different software requirements. These requirements cannot always be combined easily and sometimes even create conflicts [53], thus some virtual organizations can only use resources of specific Grid sites. This situation can be considered as a static partitioning of Grid sites with regard to virtual organizations being able to use their resources. A potential consequence of such a partitioning is low utilization on some Grid sites and overload on others, depending on the demand of the different virtual organizations. Furthermore, such a fixed partitioning cannot cope with demand surges in individual virtual organizations.

The use of virtualization can solve these problems. By running jobs inside virtual machines, they are isolated from jobs of other user running concurrently, and their data is securely stored inside the virtual machine. A malicious user must break two levels of security to compromise the provider's system and thus be able to access another user's data. Furthermore, the software requirements are always satisfied. Virtual machines can be easily transferred to any Grid site and thus resolve the dependency of virtual organizations on specific Grid sites. Finally, the isolation provided by virtualization prevents conflicts between the requirements of multiple virtual organizations using the resources of a single Grid site concurrently. This facilitates better resource sharing and thus higher utilization of physical resources.

Two approaches have been developed to combine virtualization with Grid Computing. They are briefly described in the following sections.

2.3.1 Virtual Clusters

This approach provisions entire virtual clusters consisting of a virtual head node and a set of virtual worker nodes. Each virtual head node contains both the Grid middleware that is used to submit jobs to remote Grid sites and a batch scheduler that schedules jobs to the corresponding virtual worker nodes. In this respect, a virtual cluster exhibits the same behavior and interfaces as a physical cluster, enabling the use of the same client tools for both types of clusters. This approach is depicted in Figure 2.1.

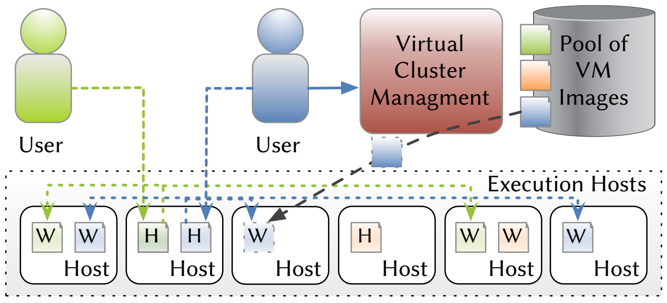


Figure 2.1 Job Execution using Virtual Clusters. This figure shows a set of Executions hosts that are used to provide three virtual clusters concurrently. The head and the worker nodes of each virtual cluster are marked with *H* and *W*, respectively. The first user (green) submits a job to his virtual cluster, whereas the second user (blue) increases the size of his virtual cluster before he submits his job.

Typically, a virtual cluster is a subset of a physical cluster and thus multiple virtual clusters can coexist on a physical cluster concurrently. The size of these clusters can be adopted dynamically to respond to demand changes, e.g., if there is a peak load in one of the virtual clusters. Consequently, virtual clusters can be considered as a way to dynamically partition a physical cluster into smaller entities. These entities are assigned to virtual organizations to provide a specifically equipped and scalable environment.

This approach is implemented in virtual clusters based on *Virtual Workspaces* [53] or *Cluster-on-Demand* (CoD) [26].

2.3.2 Dedicated Virtual Machines

Contrary to the virtual cluster approach, only a single head node running the Grid middleware and the batch scheduler is required in this approach. By integrating the management of virtual machines into the batch scheduler, the entire process of creating the virtual environment for jobs is transparent to the user. Whenever a scheduled job is going to be executed, the modified scheduler starts a copy of the virtual machine corresponding to this job on all selected nodes. Then, it executes the job inside these virtual machines. The only notable difference to a standard batch scheduler is a slight delay before the job is started. After the job is finished, the virtual machines are shut down again. This process is depicted in Figure 2.2 using the *Xen Grid Engine* (XGE) [147, 138] as an example of such an integration.

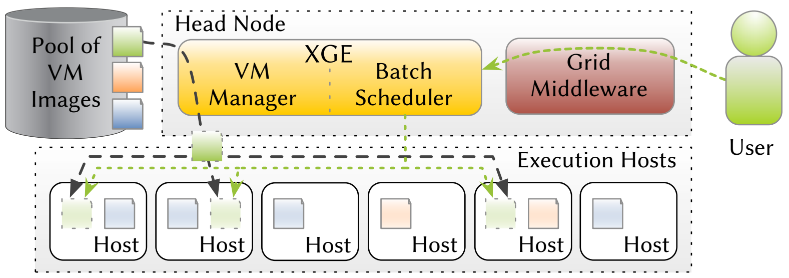


Figure 2.2 Job Execution in Dedicated Virtual Machines. This figure shows a cluster using the XGE as scheduler. A user submits his job through the Grid middleware as usual. Before the job is executed by the batch scheduling part of the XGE, it starts three copies of the corresponding virtual machine on the Execution hosts to run the job in. These virtual machines are shut down after the job is finished.

2.4 Cloud Computing

Cloud Computing is one of the biggest trends in IT today. It draws from experiences made with its ancestor Grid Computing and solves many of the remaining problems of the Grid that affect mostly commercial users. One of the advantages of Cloud Computing over the Grid is the lower barrier for using it: A user only needs a credit card to get started in the Cloud, whereas a Grid certificate and a membership in one of the virtual organizations is required in the Grid.

In the years since the term Cloud Computing first appeared, many different definitions were published. The National Institute of Standards and Technology (NIST) has published the following definition for Cloud Computing in 2011[1]:

> “ Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.
>
> *NIST [49], p. 2* ”

The NIST definition also lists five essential characteristics that can help to further distinguish Cloud Computing from the Grid: The Cloud provides *on-demand self-service*, *broad network access*, *resource pooling*, and *rapid elasticity* and is a *measured service*. Depending on the type of resources, the Grid does not have all these characteristics, e.g., for compute resources, there is neither on-demand self-service nor rapid elasticity, because these resources are assigned using a scheduler.

[1] The first version of this definition was already published in 2009.

2.4.1 Service Models

The NIST definition of Cloud Computing [49] comprises three different *services models* that provide different levels of abstraction. These models are described below starting with the highest level of abstraction and working through to the lowest level that is the focus of this thesis. With the increasing level of abstraction, the level of control over the resources, i.e., the amount of control over the complete infrastructure, is reduced. Furthermore, these service models are sometimes also called *Cloud layers*, because they additionally describe an architecture for Cloud systems in terms of layers building upon each other. The models are shown in Figure 2.3.

Software as a Service (SaaS) — This service model deals with applications or services running within the Cloud that are accessed by their users using either a browser or a specialized application. This model is thus addressed to application end users. The provider takes care of any management, deployment, or maintenance tasks regarding the application and the underlying infrastructure. Examples of the Software as a Service offers are Google Docs, Microsoft Office Online, and Dropbox.

Platform as a Service (PaaS) — This service model deals with the provision of runtime environments for applications in the Cloud. The runtime environment comprises programming languages, libraries, services, and tools. This model is thus addressed to application developers. Consequently, the user of such a runtime environment is only responsible for management and deployment of his SaaS applications, whereas the provider takes care of any management, deployment, or maintenance tasks regarding the runtime environments and the underlying infrastructure. Examples of Platform as a Service offers are the Google App Engine and Microsoft Azure.

Infrastructure as a Service (IaaS) — This service model deals with the provision of physical or virtual resources, e.g., physical or virtual machines, storage, or virtual network infrastructure. With respect to the topic of this thesis, only virtual machines are considered. It is up to the user to install an operating system and the desired applications in his virtual machines. This model is thus addressed to Cloud architects. Consequently, the user of such virtual machines is responsible for the management and deployment of the entire software stack consisting of operating system, runtime environment and application, whereas the provider is only taking care of the physical hardware the virtual machine is executed on. Examples of Infrastructure as a Service offers are Amazon Elastic Compute Cloud (EC2) and Rackspace Cloud Servers.

2.4.2 Deployment Models

The NIST definition further distinguishes between four different deployment models depending on the scope of published Cloud services [49]. A *Private Cloud* is provisioned exclusively for customers of a single organization, whereas a *Community Cloud* is provisioned for a community of customers from organizations with shared concerns

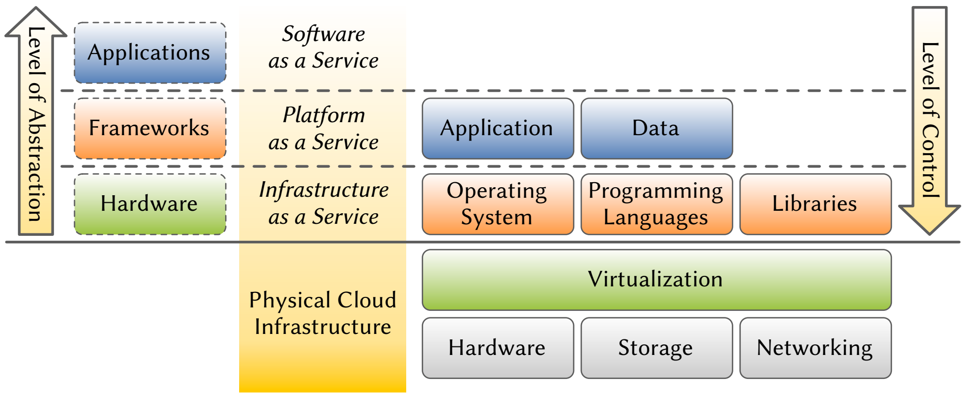


Figure 2.3 Cloud Computing Service Models. This upper part of this figure (above the solid line) shows the three different service models defined for Cloud Computing (yellow box). On the left side of the models, the level of abstraction of the particular service model is shown. On the right side of the models, the entities under control of the user in the particular service model are shown. The lower part of the figure (below the solid line) shows the physical layer, i.e., the actual hardware that is controlled solely by the provider of IaaS services.

(comparable to a Virtual Organization in the Grid). On the other hand, a *Public Cloud* is provided for use by the general public. When more than one of these models are combined, the result is called a *Hybrid Cloud*. Note that these models make no assumptions about the physical location and owner of the corresponding infrastructure, e.g., even a Private Cloud can be operated by a third party.

2.5 Virtualization

In general, virtualization is a term that denotes an abstraction of a physical resource in order to create a virtual version of that resource. The resource can be anything from network and storage devices, operating systems, to entire computer systems. Thus, virtualization facilitates running applications or operating system in a virtual environment that is independent of the specific physical computer system. This facilitates moving the virtual environment between different computer systems to respond to planned or unplanned hardware outages or overload situations. Virtualization technology can be applied at many different levels. Three examples are listed below:

Application Virtualization — This type of virtualization denotes the concept of compiling applications not for a specific hardware and operating system, but into a machine independent byte code. This byte code is then executed by a special virtual machine that provides an environment for the application. Examples of virtualization system of this type are the Java Programming Language and the .NET Framework.

Operation System Virtualization — This type of virtualization denotes the concept of running multiple logically distinct environments on a single kernel. Using different technologies, the access of the environments to the file system, hardware, and status information is restricted. The amount of such environments that can be active concurrently is much higher compared to the number of concurrently active virtual machines using machine virtualization. Examples of virtualization systems of this type are *chroot jails* and Linux VServer.

Machine Virtualization — This type of virtualization denotes the concept of providing virtual versions of entire computer systems (virtual machines) that run of-the-shelf operating systems (with a few exceptions described below). This technology facilitates running different operating systems concurrently on a single computer system and provides strong isolation between them. Examples of virtualization systems of this type are given below.

The focus of this thesis is machine virtualization, i.e., the virtualization of entire computer systems that is used in both Virtualized Grid and Cloud Computing (IaaS). In the following sections, a formal definition of virtualization is presented in the context of machine virtualization, and different virtualization approaches for virtual machines on the x86 architecture are described.

2.5.1 Formal Definitions

Popek and Goldberg have defined formal requirements for virtualizable architectures in 1974 [117]. They proposed the idea of a *Virtual Machine Monitor* (VMM) or hypervisor:

> “ A virtual machine is taken to be an *efficient, isolated duplicate* of the real machine. We explain these notions through the idea of a *virtual machine monitor* (VMM). [...] As a piece of software a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.
>
> *Popek and Goldberg [117], p. 413* ”

The meaning of these three characteristics of a hypervisor is described below [117]:

Equivalence — The first characteristic (denoted by the term “essentially identical” in the quote) means that a program running under the VMM should exhibit a behavior that is essentially identical to the behavior of the same program when running on the original machine. The constraint expressed by “essentially” refers to differences caused by timing dependencies and the availability of system resources.

Efficiency — The second characteristic means that a statistically dominant subset of machine instructions must be executed directly on the processor, without the VMM intervening.

Resource Control — The last characteristic means that the VMM is in complete control of the resources, e.g., memory or peripherals. A program running under the VMM may not access resources that are not allocated to it by the VMM. Furthermore, the VMM might revoke allocations under certain circumstances.

Based on the definition of the Virtual Machine Monitor above, Popek and Goldberg define the term *Virtual Machine* as follows:

> “ A *virtual machine* is the environment created by the virtual machine monitor.
>
> *Popek and Goldberg [117], p. 413* ”

To determine whether a computer architecture is virtualizable, Popek and Goldberg classify its instructions. Their work is based on a third generation computer, i.e., computers based on integrated circuits[2] that were built roughly in the second half of the 1960s. One particular feature that distinguished a third generation computer from it predecessors was *virtual memory* in the form of segmentation with or without paging [35]. Virtual memory provides memory protection, i.e., a process can only access information in its own address space, but not in the address space of other programs or the operating system. This is enforced using a hardware assisted mapping mechanism.

A related feature is the distinction between two modes of operation: the *supervisor mode* and *user mode*. Memory protection can only be enforced in combination with these modes of operation: the data structures controlling the memory mapping, e.g., base (or relocation) and bounds registers, may only be modified in the supervisor mode. Without this distinction, a process could overwrite the data structures and consequently access any memory location.

For this kind of computer, Popek and Goldberg came up with the following classification [117]:

Privileged Instructions — A privileged instruction is an instruction that *traps* when it is executed in user mode, whereas it is executed in supervisor mode. An instruction is said to trap if the processor executes a specified routine (of the operating system) instead of the instruction. This is called a privileged instruction trap. Note that an instruction that is merely skipped in user mode, but does not trap, is not a privileged instruction with regard to this definition.

Sensitive Instructions — This class of instructions that is of particular importance for virtualizable architectures can be further subdivided:

Control Sensitive Instructions — A control sensitive instruction is an instruction that modifies either the mode of operation the processor runs in, both from user to supervisor or vice versa, or the memory mapping – and thus the amount of available resources.

Behavior Sensitive Instructions — A behavior sensitive instruction is an instruction that has different effects on the state of the processor depending

[2] In contrast to transistor or vacuum tube based computers from the second and first generation, respectively [35].

on the mode of operation the instruction is executed in – supervisor or user mode – or depending on the location of the instruction.

Innocuous Instructions — An innocuous instruction is an instruction that is not sensitive.

Based on this classification, Popek and Goldberg establish the following theorem:

> “ For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.
>
> *Popek and Goldberg [117], p. 417* ”

This theorem ensures that only the hypervisor (running in supervisor mode) is able to allocate resources to virtual machines (running in user mode) and that there is no way for a virtual machine to increase their privileges, i.e., switch to supervisor mode. Both would require a sensitive instruction that is by definition privileged and causes a trap to the hypervisor when executed by a virtual machine running in user mode. The hypervisor can then emulate the functionality of the sensitive instruction and return control to the virtual machine (skipping the sensitive instruction). This is called *trap and emulate* and the fundamental concept of virtualization, because it enables the hypervisor to stay in control of the system, to isolate virtual machines from each other, and to maintain the virtual machine's illusion of full control over the computer for the virtual machine. On the other hand, innocuous instructions are executed directly on the processor, because they have no effect on either the hypervisor or other virtual machines.

In his thesis, Goldberg also classifies virtual machine monitors into two different types [60]. These types are depicted in Figure 2.4.

Type I VMM — A Type I VMM runs directly on the hardware. It is also called *native* or *bare-metal* hypervisor.

Type II VMM — A Type II VMM runs under a host operating system. It is also called a *hosted* hypervisor.

In order to distinguish the operating system a Type II hypervisor runs on from the operating system that runs in a virtual machine created by that hypervisor, the operating systems are called *host operating system* and *guest operating system*, respectively. The term guest operating system is also used with Type I hypervisors, although there is host operating system to distinguish it from when a Type I hypervisor is used.

Each of those types of hypervisors has its own advantages and disadvantages. Type I hypervisors are said to offer better performance because they operate directly on the hardware without relying on an operating system, but there are no reliable sources to support this assumption. On the other hand, a Type I hypervisor needs to fully to support the hardware it runs on, i.e., it needs drivers developed for this hypervisor. Typically, this leads to a limited set of supported hardware in comparison to a Type II VMM that uses the host operation systems and its drivers to access the hardware.

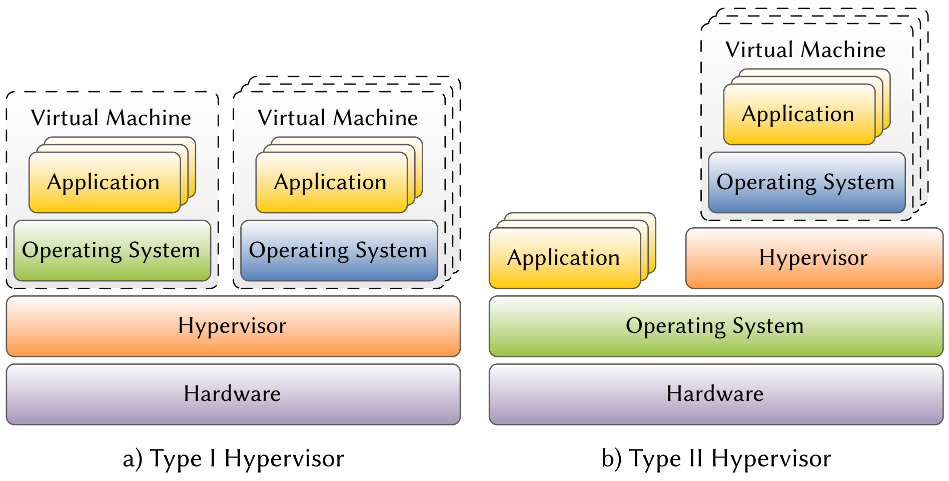


Figure 2.4 Types of Hypervisors. This figure depicts the differences between Type I and Type II hypervisors as defined by Goldberg.

2.5.2 Virtualization Approaches for the x86 and x86-64 Architectures

Up to the middle of the last decade, the processors of the x86 and the newly introduced x86-64 Architectures did not satisfy the criteria for virtualizable architectures defined by Popek and Goldberg [117]. Robin and Irvine [129] evaluated the Intel Pentium architecture with regard to its ability to support a virtual machine monitor in 2000. They first verified that the architecture still meets the assumptions about the design of a computer that Popek and Goldberg based their work on, e.g., two modes of operation and memory protection. Then, they reviewed the instruction set, found 17 critical instructions and concluded:

> “ The Intel architecture uses interrupts and traps to redirect program execution and allow interrupt and exception handlers to execute when a privileged instruction is executed by an unprivileged task. However, the Pentium instruction set contains **sensitive, unprivileged instructions**. The processor will execute unprivileged, sensitive instructions without generating an interrupt or exception. Thus, a VMM will never have the opportunity to simulate the effect of the instruction.
>
> *Robin and Irvine [129], p. 5* ”

Consequently, the first virtualization products that appeared for the x86 Architecture in 1999 had to use *binary translation* to deal with nonvirtualizable instructions [160]. Starting in the years 2005 and 2006, Intel and AMD sold the first x86-64 processors that satisfied the aforementioned criteria with the Intel VT-x and AMD-V extensions, respectively. The first generation of these processor extensions did only cover the CPU itself, which resulted in poorer performance compared to the existing binary translation in many circumstances [3].

Newer processors released in 2008 and 2007 include the Intel Extended Page Tables and AMD Rapid Virtualization Indexing extensions, respectively. These extensions provide support for second level address translation or memory virtualization and improve the performance of virtualization when the processor extensions are used. Since then, the hardware support for virtualization has been further extended. Current hardware also supports virtualizing the interrupt controller, external devices, and graphics processors.

As a result of the missing support for virtualization in earlier processors, two different virtualization approaches are used on the x86 and x86-64 architectures that are described in the next two sections.

2.5.2.1 Full Virtualization

The full virtualization approach virtualizes a system in a way that is transparent to the guest operating system. The advantage of this approach is that the guest operating system does not need to be modified in order to run as virtual machine created by a hypervisor using this approach. To reach this goal, a complete computer system needs to be emulated, e.g., hard disks, network devices, and video cards, but also low-level device like a system chipset or a virtual BIOS [160]. With the increasing hardware support for virtualization, the amount of devices that need to be emulated will likely be reduced in the future.

As stated in the last section, a hypervisor has to trap and emulate sensitive instructions, whereas innocuous instructions can be executed directly on the processor without intervention of the hypervisor. The are two techniques to implement full virtualization on the x86 architecture depending on the availability of hardware support for virtualization, or more specifically the trapping of all sensitive instructions.

Full Virtualization with Binary Translation — The binary translation approach was developed by VMware that introduced virtualization to the x86 architecture in 1999. Kernel code is translated by the hypervisor to replace nonvirtualizable, i.e., sensitive, instructions with new sequences of instructions that have the intended effect on the virtual hardware. The translation of kernel instructions is done on the fly and the result of the translation is cached for future use. User level code, on the other hand, is executed directly on the processor and runs at native speed [160].

This type of full virtualization can be found for example in *VMware ESXi* or *Workstation* [159] or *VirtualBox* [114].

Hardware Assisted Full Virtualization — With the introduction of hardware support for virtualization in the x86 architecture by Intel and AMD, it is no longer necessary to use binary translation to deal with sensitive instructions. The hypervisor runs in a new privilege level and all sensitive instructions trap to it if they are not executed in the new privileged level. Moreover, the hypervisor can even influence which instructions cause traps using a special control structure that is used by the processor to store the state of a virtual machine. Using the hardware extensions, a classical trap and emulate hypervisor can be implemented on the x86 architecture.

The type of full virtualization can be found in all virtualization solutions mentioned above and additionally in *KVM* [79] and *Xen* when using *Xen Hardware Virtual Machine* (HVM) guests [174].

2.5.2.2 Paravirtualization

The paravirtualization approach virtualizes a system in a way that is not only visible to the guest operating system, but also requires the guest operating system to be modified. This approach supports virtualization even on nonvirtualizable processor architectures. Every nonvirtualizable instruction in the guest operating system's kernel needs to be replaced with a so-called *hypercall*, i.e., a software trap to the hypervisor[3] [13]. In contrast to the full virtualization approach, the guest operating system has to cooperate with the hypervisor for the paravirtualization approach to work.

The major advantage of this approach is the low overhead resulting in the use of efficient hypercalls instead of expensive instruction trapping. The major drawback, on the other hand, is the need to modify the kernel of each operating system that will be run in a virtual machine created using this virtualization approach. This prevents the use of proprietary and legacy operating system. Another drawback is the awareness of the guest operating system about running in a virtual machine.

The most prominent hypervisor implementing the paravirtualization approach is Xen when using *Xen Paravirtualization* (PV) guests [174]. Its architecture is depicted in Figure 2.5). Other implementations of this approach were the Denali Isolation Kernel [168] and the VMware Virtual Machine Interface (VMI) [9].

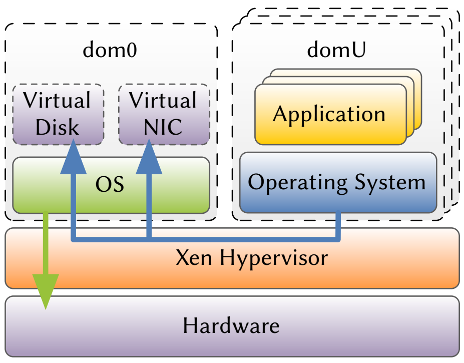


Figure 2.5 The Xen Hypervisor Architecture. This figure shown the architecture of the Xen hypervisor and emphasizes the key role played by the dom0 both for accessing the actual hardware and for providing virtual disks and network interfaces for the unprivileged domains.

A particular feature of the Xen hypervisor is the use of a privileged virtual machine or *domain* in the Xen nomenclature. This privileged virtual machine is called *dom0* (domain zero), because it is the first domain that is started directly after the hypervisor. It is responsible for running the Xen daemon and administrative software to control

[3] The name hypercall is a pun on syscall that is a software trap to the operating system.

the hypervisor. Furthermore, it has access to the hardware, contains the corresponding drivers and provides virtual network interface controllers (NICs) and block devices for the other virtual machines running on the hypervisor. These virtual machines are called *domU* (unprivileged domain) to distinguish them from the privileged one.

Device drivers are another use case for the paravirtualization approach. Hypervisors implementing the full virtualization approach typically provide paravirtual drivers for the emulated devices, especially for hard disks, network devices, and video cards. These drivers significantly improve the performance of virtual machines, because they bypass the emulation logic required for accessing these devices with regular drivers [101].

2.6 Summary

In this chapter, Virtualized Grid and Cloud Computing were presented. These are the environments the solutions proposed in this thesis are designed for. Additionally, the basic concepts of Grid Computing, the predecessor of Virtualized Grid Computing, were introduced. Finally, virtualization in general and virtualization approaches for the x86 architecture that is prevalent in both desktop and server systems were described.

This page is intentionally left blank.

"There is no harm in repeating a good thing."

Plato (ca. 427–347 B.C.)

3 Lifecycle Management

3.1 Introduction

The appearance of virtualization technology on the Intel x86 architecture in the late 1990s caused an increasing trend of virtual machine usage. Starting with virtualization products for workstations that were used for easing operating system migrations and provisioning of testbed systems for developers, virtualization technology made its way to servers. At the beginning, the objective was primarily server consolidation: physical servers are replaced by virtual servers and these virtual servers are consolidated to run on fewer physical machines. Later on, other objectives came into focus. Virtual machines made the isolation of different applications for security reasons possible without relying on additional hardware. Migration techniques enabled administrators to react to overload situations by moving virtual machines to other execution hosts. *Virtual Appliances*, the bundling of software and its execution environment into a virtual machine, emerged as a new distribution mechanism for software.

Besides the technological innovations described above, virtualization technology also prompted improvements in other areas like usability. Grid Computing was introduced to make compute and storage resources of computing centers available for external users. It tried to overcome the usability problems of heterogeneous compute clusters by introducing an overarching, certificate-based authentication scheme for users and unified interfaces for interacting with different scheduling and storage systems. Virtualized Grid Computing solved one of the remaining problems of Grid Computing: the availability of required software at remote computing centers. It applied the concept of Virtual Appliances to Grid Computing. Users no longer only submitted a description of a job including the required data to a computing center, but also a virtual machine containing the required programs.

At this stage, most of the technical problems were solved. Cloud Computing builds upon all of those developments, combined them with a business model, and changed the focus to service orientation. It enables users all over the planet to self-provision virtual infrastructures on-demand using customized virtual machines[1]. One on the main advantages of a virtual infrastructure over physical hardware is the virtual infrastructure's adaptability, i.e., the possibility to scale the infrastructure up or down with almost no delay. This flexibility makes Cloud Computing an ideal approach for each application with hugely varying workloads or as a fallback for overload situations.

Some of the features of virtualization technology itself, but especially the way virtual machines are used in Virtualized Grid and Cloud Computing, radically change the lifecycle of virtual machines compared to physical machines. In the remainder of this chapter, the lifecycle of virtual machines and its implications for the management of virtual machines are examined.

3.2 Related Work

3.2.1 Lifecycle of Virtual Machines

Most publications that mention the term lifecycle in the context of virtual machines use it to refer to the entire lifetime of a virtual machine, without going into details about what happens during that period. The only publication that deals with the lifecycle of virtual machines is a description of security challenges in virtual environments by Garfinkel and Rosenblum [59]. They observed that the state of a machine, which is typically visualized as a straight line, is comparable to a tree in case of a virtual machine. The reason is the snapshot feature of many hypervisors allowing users to capture the state of their virtual machines at any point in time. In principle, taking snapshots has no influence on the lifecycle of a virtual machine. However, snapshots are typically used to roll virtual machines back to earlier states in order to recover from errors. Furthermore, snapshots facilitate cloning of virtual machines and thereby creating new branches in the state tree. Garfinkel and Rosenblum describe the effects of this change of the state model with regard to the security of the virtual machines and the communication with them. In contrast to this thesis, Garfinkel and Rosenblum limit the term lifecycle to changes of a virtual machine's state, which is characterized by configuration changes, installation or updating of software, and the execution of software. This thesis takes a broader view on the lifecycle of virtual machines and also examines phases in which the virtual machine is not running at all.

3.2.2 Virtual Machine Management Systems

According to the definition of Popek and Goldberg, a hypervisor is responsible for creating an environment for running a guest operating system. Therefore, it is a central component for virtualization, although it is typically not enough to manage virtualized infrastructures. A virtual machine management system is the interface

[1] Cloud Computing is used as a synonym for the Infrastructure as a Service model.

to a hypervisor that allows users to deal with virtual machines, e.g., to start, stop, or monitor virtual machines or to manage virtual machine images.

Many commercial virtualization products targeted at the desktop, e.g., VMware Workstation [159] or VirtualBox [114], already include a virtual machine management system in addition to the plain hypervisor. These management systems are obviously limited to the features provided by the desktop class hypervisors. Apart from the basic operations on virtual machines, i.e., starting, stopping, pausing, or resuming virtual machines, they typically facilitate the creation of snapshots, cloning of virtual machines, as well as importing and exporting of virtual machines from and to the *Open Virtualization Format* (OVF) [36], respectively, to enable exchanging of virtual machines between different hypervisors.

Server class hypervisors, e.g., KVM [79], Xen [174], or VMware ESXi [159], typically provide more advanced features, e.g., live migration[2]. or high availability. Many of these hypervisors are Type I hypervisors and thus they typically do not have an integrated management system, but are administrated via network. Multiple virtual machine management systems are available and many of these are based on *libvirt* [21], a library that provides a unified interface to control many different hypervisors. Examples of such systems are *virt-manager* [158], *oVirt* [115], and the proprietary *VMware vCenter*[3] [159].

In addition, the management Dashboards and APIs of Infrastructure as a Service Clouds are virtual machine management systems. This applies both to commercial offers like Amazon Elastic Compute Cloud (EC2) and to Open Source Cloud systems like *OpenStack* [111], *Eucalyptus* [107], and *Nimbus*[4] [53].

We have previously developed the *Image Creation Station* (ICS) [43] and *Xen Grid Engine* (XGE) [147, 43] in our effort to promote Virtualized Grid Computing. Together, they also form a virtual machine management system. Additionally, we have previously developed tools for *Grid Workflows* that encourage hosting services in virtual machines: an automatic failover system that responds to infrastructural failures, e.g., network timeouts and server outages, by switching to a backup service or by deploying a virtual machine containing the service into the Cloud [76] and an extensible workflow editor for creating such workflows [38].

3.3 Lifecycle of Virtual Machines

Virtual machines can be easily created on-demand and destroyed when not needed any longer. Using a virtual machine management system, a new virtual machine is literally just a click away, in contrast to the typically much longer acquisition and deployment times required before a new physical machine can be used.

When not used, virtual machines can be stopped or suspended, leaving nothing more

[2] VirtualBox also supports live migration, although it is not a typical server class hypervisor.
[3] VMware vCenter is restricted to controlling VMware products and thus not based on libvirt.
[4] Nimbus emerged from Virtual Workspaces, an implementation of Virtualized Grid Computing. It has been extended to provide a complete IaaS Cloud system.

than a disk image and, in the latter case, state information. Thus, non-running virtual machines – also called *dormant virtual machines* [167] – induce almost no cost, except for storage. This enables new usage models that are not possible without virtualization: a virtual machine can be created, prepared for usage and then shut down, staying dormant until it is needed. At this point, it is started again and executed until it is eventually shut down again, and the process is repeated. This approach to dynamically adapt the number of running virtual machines to the workload is called on-demand provisioning [69], and it is one of the key selling points of Cloud Computing.

As a consequence, there is no uniform virtual machine lifecycle, but different lifecycles depending on the individual usage model. The usage models can be categorized based on the mode of execution of the virtual machines as follows:

Continuous Execution Model — Virtual machines are executed as long as they are needed, even in periods without active usage. This usage model is similar to the typical usage model of physical machines.

On-demand Execution Model — Virtual machines are started on-demand and executed as long as they are actively used. During periods without active usage, they are dormant. There is no corresponding usage model of physical machines, since the cost of a dormant physical machine (unused hardware) is much higher compared to that of a dormant virtual machine (storage of a disk image and state information).

These models, the resulting virtual machine lifecycles and their implications will be described in more detail in the two following sections.

3.3.1 Continuous Execution Model

This execution model is often found when virtualization is used to consolidate multiple physical machines, by converting them to virtual machines and executing them on shared resources to reduce the amount of hardware required. Applications calling for the continuous execution model are typically individual servers that are the *point of contact* for client requests[5]. Examples of such servers are web, email and file servers that are expected to be available at all times.

The goals of this model are reducing the costs for hardware, energy, and cooling and improving manageability. The former is accomplished by reducing the number of physical machines and optimizing resource utilization. The latter is accomplished by features provided by advanced virtualization techniques, i.e., migration of virtual machines to distribute the load more evenly between physical machines. Scalability, on the other hand, is not a goal that can be accomplished with this execution model[6]. Like with physical machines, virtual machines have to be provisioned for the predicted peak load (peak load provisioning) [44] when this execution model is used. When scalability is an issue, a more dynamic approach is required to adjust the provided

[5] In contrast to servers that are "hidden" from the client by load balancers or similar systems.

[6] Except from migrating virtual machines to faster or less used physical machines to improve their performance.

computing power to match the request volume: an on-demand execution model, described in Section 3.3.2, can achieve this flexibility.

3.3.1.1 Lifecycle

The lifecycle of a continuously executed virtual machine, shown in Figure 3.1, obviously is not a cycle, but a sequence of 5 phases, starting with the creation of the machine and ending with its deletion. Nevertheless, the term lifecycle is used in the remainder of this section. The phases are described in detail below.

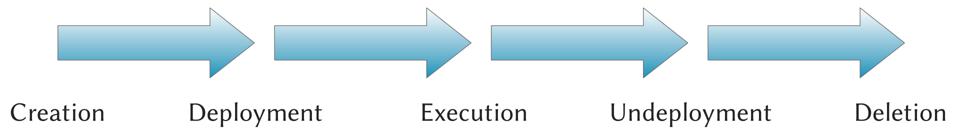


Figure 3.1 Lifecycle of a Virtual Machine in the Continuous Execution Model. In this model, a virtual machine has a lifecycle consisting of 5 sequential phases: *Creation* and *Deployment*, the actual *Execution* (including maintenance) as well as *Undeployment* and *Deletion*.

Creation Phase — In the creation phase, the virtual machine is created using a suitable management tool. This phase typically involves creating an empty disk image as well as installing and configuring the operating system and applications. The process of installing the operating system can either be automated or performed manually, i.e., requiring interaction with the user creating the virtual machine. Installing and configuring the required applications is obviously a manual task.

A template can be used to speed up the creation of virtual machines instead of creating every new virtual machine from scratch. Depending on the management tool, these templates are named differently: *Amazon Machine Image (AMI)* in case of Amazon Elastic Compute Cloud [8] or *golden image* in case of the Image Creation Station [43]. It contains a pre-configured operating system installation, but no applications except for common tools. The golden image is cloned and its copy is used as a disk image for the new virtual machine.

Deployment Phase — In the deployment phase, the virtual machine is prepared for execution. This mainly involves copying its disk image to the execution host, unless the disk image is stored on a network file system, and starting it. Additional pre-execution tasks can also be part of this phase, i.e., adopting the configuration of the virtual machine to the environment or creating temporary disk images used as scratch space.

Execution Phase — The execution phase, the most prevalent of the 5 phases, is the productive part of the virtual machine's lifecycle. In this phase, the virtual machine is doing whatever is necessary to satisfy its intended purpose, e.g., serving files, emails or web content. The virtual machine stays in this phase even in periods without utilization, until it is no longer needed and therefore retired.

An important task during the execution of the virtual machine is software maintenance, i.e., checking for available updates and installing updates if necessary. As with physical machines, continuous software maintenance is required throughout the execution phase, to keep the virtual machine, the applications as well as the data used by the virtual machine safe. The same holds true for security monitoring of the running virtual machine, to detect anomalies and respond to attacks.

Undeployment Phase — This phase is the preparation of the retirement of the virtual machine. It is shut down and any temporary disk images belonging to the virtual machine are deleted from the execution host. The disk image of the virtual machine is also deleted from the execution host, unless it is stored on a network file system.

After this phase, the virtual machine could theoretically be deployed to an execution host again, and the lifecycle would continue in the deployment phase. But in this case, the on-demand execution model describes the lifecycle of the virtual machine better, so redeployment is out of scope for the continuous execution model.

Deletion Phase — In this last phase, the virtual machine itself and its disk image are ultimately deleted using a management tool.

The lifecycle of a continuously running virtual machine with its 5 phases is very similar to that of a physical machine. Instead of being created, physical machines are acquired, but otherwise the first phase is identical. The deployment phase consists of physically moving the machine to its operating site, instead of moving disk images to an execution host. The same applies to the undeployment phase. Finally, physical machines are either sold or disposed instead of deleted. The last two phases are sometimes combined to a single phase called *retirement* phase.

3.3.1.2 Implications

The last section highlighted the similarities between the lifecycle of virtual machines in the continuous execution model and the lifecycle of physical machines. These similarities are not surprising, since the goal of this model is to replace physical with virtual machines. As a result, the same maintenance methods and tools used for physical machines during the execution phase can also be used for their virtual counterparts. For security monitoring, however, novel tools that are aware of the virtualization are required. Using this knowledge, such monitoring tools can provide more meaningful insights into what is happening in a virtualized system. For the other phases, only basic tooling for virtual machine management is required, which supports the fundamental operations with virtual machines, i.e., creation, deployment, starting, stopping, undeployment, and deletion of a virtual machine.

3.3.2 On-demand Execution Model

In the following, two types of applications are presented that benefit from another, more dynamic execution model. The first type consists of modern web applications, such as websites or online shops. One of the problems web applications face today are heavy fluctuations in the amount of concurrent visitors. Such fluctuations can be triggered by different events, i.e., coverage on popular news sites in case of a web site [4, 40] and the holiday season or special shopping events like Cyber Monday in case of an online shop[7]. In both cases, significantly more visitors are attracted by a site and the workload on the servers increases.

Independently of being run in a virtualized environment, modern web applications typically use load balancers that distribute the workload equally between a number of web servers. These load balancers decouple the point of contact from the actual machine that serves a request and thus enable the use of dynamically provisioned resources. Using the on-demand execution model, new virtual machines can be created on the fly (scale up) if the workload cannot be handled by the existing virtual machines. If the workload reduces at a later time, some of the virtual machines can be shut down (scale down). This is called elastic scaling [44] and is one of the key features of modern Cloud architectures. The on-demand execution model can thus handle workload fluctuations better than any static approach, i.e., physical machines or virtual machines in the continuous execution model that relies on peak load provisioning to cope with fluctuating workloads.

The second type of applications benefiting from the on-demand execution model consists of scientific applications that require large-scale computing systems. In many scientific fields, computing is widely used as a tool, typically in the form of programs executed on compute clusters in the universities' computing centers. The need for this kind of computing also exists in the commercial world, as experiences from the PT-Grid, FinGRID, and Biz2Grid projects have shown. Virtualization and the on-demand execution model can be applied to this usage scenario in different ways.

Virtualized Grid Computing tries to improve the *usability* of compute clusters in the face of rapidly changing software requirements. It allows users to not only submit a job using the scheduling system of a cluster, but also provide a virtual machine that contains the required software in a working environment. Users are thus no longer depending on administrators to install the software they need on the compute cluster. This also allows the execution of jobs on remote compute clusters without checking for the existence of the required software beforehand. The latter was one goal of the Grid initiatives worldwide: allow the remote use of universities' compute clusters, either when a local cluster was not available, not powerful enough or in use by others.

The use of virtual machines is a foundation of Cloud Computing, thus the usability problem of compute clusters solved by Virtualized Grid Computing is solved per

[7] Amazon's CTO wrote on his blog that they invented *Dynamo*, Amazon's custom key-value storage system, because of-the-shelf storage and database solutions could not keep up with the load during the 2004 holiday shopping season (apparently, they could during the remaining time of the year). This caused several outages [161].

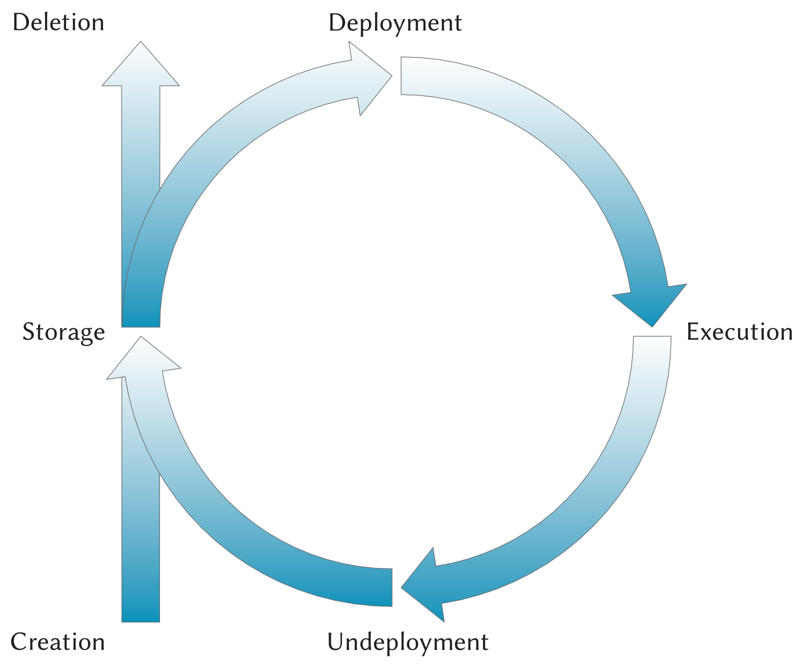


Figure 3.2 Lifecycle of a Virtual Machine in the On-demand Execution Model. In this model, a virtual machine has a lifecycle consisting primarily of 4 phases: *Storage*, *Deployment*, *Execution*, and *Undeployment*. Two additional phases mark the start and the end of the virtual machine's life: *Creation* and *Deletion*.

se. Using Cloud Computing, a complete computing environment can be provided on-demand, i.e., a virtual compute cluster consisting of a group of virtual machines that exists only for the duration of a program's execution (days, weeks or sometimes even months) [77]. Another feasible approach is to combine existing resources like a compute cluster with virtual machines in the Cloud, started on-demand to temporary increase the computing power to satisfy computation requirements [37]. Again, the on-demand execution model can solve the scalability problem.

3.3.2.1 Lifecycle

The lifecycle of a virtual machine in the on-demand execution model is shown in Figure 3.2. In contrast to the lifecycle of virtual machines in the continuous execution model, presented in Section 3.3.1, it is a real cycle with an entry as well as an exit phase, namely the creation and deletion of the machine. The phases are described in detail below.

Creation Phase — In the creation phase, the virtual machine is created using a suitable management tool. This phase is identical to its counterpart in the continuous execution model. A detailed description of this phase can be found in Section 3.3.1.1.

Storage Phase — In the storage phase, the virtual machine is being kept in dormant state in an appropriate *storage system*. Although ultimately depending on the individual use case, this is likely the most prevalent phase in the lifecycle.

Deployment Phase — In the deployment phase, the virtual machine is prepared for execution, i.e., copied from the storage system to the execution host, if required, and started. A detailed description of this phase can be found in Section 3.3.1.1.

Execution Phase — The execution phase is the productive part of the virtual machine's lifecycle. In this phase, it is executed on the *execution host* and serving its intended purpose, e.g., serving files, emails or web content, solving scientific problems or doing simulations. Unlike in the continuous execution model presented in Section 3.3.1.1, the virtual machine stays in this phase only in periods of active usage, and is shut down immediately if it is no longer needed. Thus, the execution phase is not necessarily the most prevalent phase in the lifecycle of a virtual machine anymore. However, the need for continuous security monitoring of virtual machines during the execution phase is not lessened by the on-demand execution model.

Undeployment Phase — After the execution phase, the virtual machine is shut down and any temporary disk image belonging to the virtual machine is deleted. Depending on the individual case, it is necessary to copy the disk image of the virtual machine back to the storage system to make changes made during the execution phase persistent.

Deletion Phase — In this phase, the virtual machine that is no longer running, but only kept in dormant state in the storage system, is deleted.

Cloning of Virtual Machines

The concept of cloning virtual machines is used very often in the on-demand execution model. When a number of virtual machines are provisioned, a single virtual machine is deployed to all of the selected execution hosts, i.e., multiple copies or clones of the corresponding disk image are created. This approach is used when multiple identical virtual machines are required as worker nodes, e.g., when preparing the execution environment for a job in Virtualized Grid Computing or when provisioning additional web servers in the Cloud to scale up a web application.

These cloned virtual machines are typically considered as temporary virtual machines and thus deleted immediately after usage. In terms of the lifecycle presented in Figure 3.2, they are not written back to the storage system during the undeployment phase, and the original virtual machine is not deleted from the storage system during the deletion phase.

An important implication of virtual machine cloning is related to the location of data. Obviously, data cannot be stored in a virtual machine, because any change would be lost after the cloned virtual machine is deleted. Thus, external systems need to be used to store the data required by the application. This requirement is not related to the use of virtual machines, but affects distributed applications in general.

Maintenance of Virtual Machines

The main differences that distinguish the on-demand execution model from the continuous execution model are the repeated deployment – execution – undeployment cycles and the frequent, potentially longer-term storage phases, during which the virtual machine is dormant and kept in the storage system. Especially the storage phases restrict the possibilities to do required software maintenance tasks. Contrary to the continuous execution model, the execution phase is no longer the ideal phase for software maintenance, since a virtual machine might not be in the execution phase when software maintenance is required, i.e., a security update needs to be installed, but instead be dormant and kept in the storage system. After the next deployment phase, this virtual machine might not be up-to-date.

One potential solution to this problem is to schedule software maintenance at the end of the deployment phase, after the virtual machine has been started. While this approach guarantees that the virtual machine is up-to-date at the beginning of the execution phase, it prolongs the time required to scale up, because the just started virtual machine cannot be used immediately, but only after all maintenance tasks are finished. This is unfortunate both for modern web applications, as it delays the effect of the scale up and thus the performance improvement, and scientific applications, as it reduces the amount of time available for solving the actual problem.

Another potential solution is software maintenance at the beginning of the undeployment phase, before the virtual machine is shut down. This approach does instead prolong the time to scale down, which is less critical if enough execution hosts are available, e.g., in the Cloud. In the case of Virtualized Grid Computing, it delays the execution of the next job. Even worse, it is a questionable approach, because there is no guarantee that the virtual machine is still up-to-date when it is started the next time.

Since the execution phase is not suited for software maintenance in the on-demand execution model, the prevalent storage phase seems to be a better opportunity for software maintenance. A well-maintained virtual machine is always up-to-date after deployment and can be used immediately. Software maintenance of dormant virtual machines also aligns perfectly with the concept of cloning, i.e., starting multiple virtual machines using the same disk image. All clones are up-to-date if the maintenance is done during the storage phase, contrary to the approach of scheduling the software maintenance at the end of the deployment phase described above. The latter would require each clone to be maintained individually. An updated version of the lifecycle including maintenance is shown in Figure 3.3 and a description of the newly added maintenance phase is given below.

Continuous Maintenance Phase — In the continuous maintenance phase, a dormant virtual machine is continuously monitored by a specialized management system. The monitoring does not only include the important task of checking for the availability of updated software, which is automated in most operating systems but obviously not working for dormant virtual machines. Additional scans using established vulnerability scanners should be performed to detect

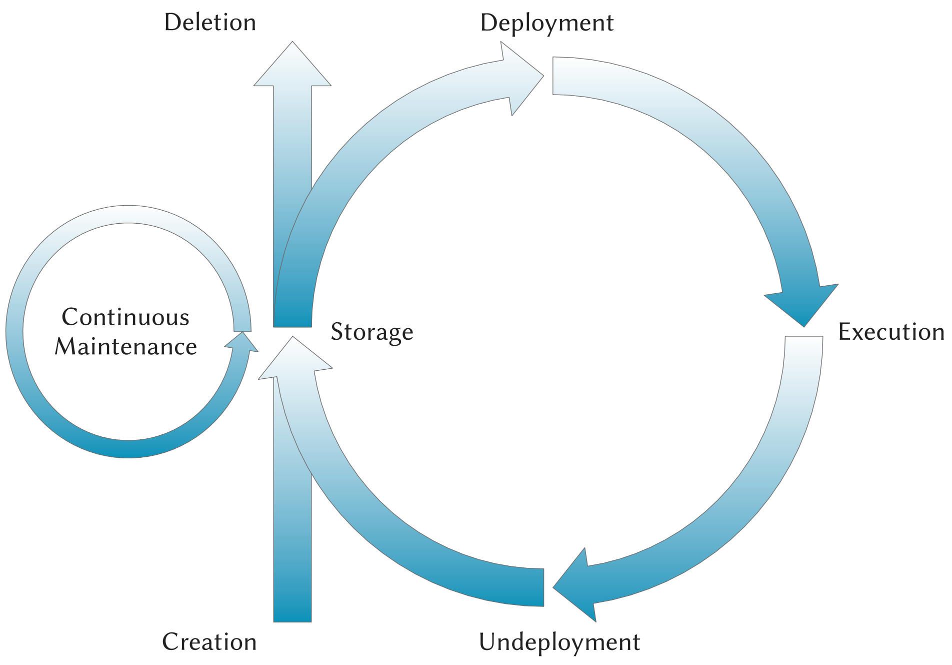


Figure 3.3 Revised Lifecycle of a Virtual Machine in the On-demand Execution Model. In this model, a virtual machine has a lifecycle consisting primarily of 4 phases: *Storage*, *Deployment*, *Execution*, and *Undeployment*. Two additional phases mark the start and the end of the virtual machine's life: *Creation* and *Deletion*. During the *Storage* phase, each virtual machine is subject to *Continuous Maintenance* to guarantee it is up-to-date at boot time.

vulnerabilities created by configuration errors and vulnerable software, for which no update has yet been published. Available updates should be installed automatically, if they do not break they system, which is the case for most security updates.

Note that software maintenance in the storage phase does not supersede maintenance in the execution phase in all cases. Especially for web applications, software maintenance even during the execution phase is of vital importance if vulnerabilities are found. Fortunately, the load balancer hides the individual virtual machines that handle the workload of the application, so an outdated virtual machine can be replaced with an up-to-date one without any outages.

3.3.2.2 Implications

The on-demand execution model has some implications for the handling of virtual machines and virtual machine images. These implications can be categorized into the following phases in the lifecycle shown in Figure 3.3:

Deployment Phase — In contrast to the continuous execution model, the deployment of a virtual machine is a frequent process. Furthermore, the time required to scale up is dominated by the time required to deploy a virtual machine to an execution host. Optimizations of the deployment process thus improve the elasticity of the overall system, which is a performance indicator directly perceivable by customers and one of the main reasons to use Cloud infrastructures in the first place.

An additional requirement results from cloning virtual machines to deploy multiple virtual machines. While it is technically possible to launch multiple virtual machine using a single, unmodified disk image, it requires careful preparation of the virtual machine and external services to configure the individual cloned virtual machine correctly. Mechanisms for configuring the virtual machine during deployment would simplify the usage of the virtual machine cloning approach.

For the stated reasons, it is important for Virtualized Grid and Cloud Computing providers to optimize the deployment time of their infrastructure. Solutions that improve the deployment time and provide a configuration facility are described in Chapters 4 and 5.

Storage Phase — In contrast to the continuous execution model, virtual machines in the on-demand execution model can be dormant for arbitrary periods of time and thus need to be stored in a storage system. A dormant virtual machine typically consists of nothing more than a disk image that contains the operating system and installed applications. Such disk images are traditionally stored as plain files on standard file systems. Unfortunately, this kind of storage does not scale well for large numbers of virtual machines [128] that are the result of a widespread adoption of Virtualized Grid and Cloud Computing approaches. An efficient storage system for virtual machines is required to keep pace with the trend of increasing numbers of virtual machines. To lay the foundation for large virtual machine archives, it might be necessary to retire full disk images as the storage format for virtual machines and explore new techniques like decomposition, compression or deduplication for storing virtual machines.

Improved storage formats and systems for virtual machine images are described in Chapters 4 and 5. They do not only solve the storage related problems of large numbers of virtual machines, but also improve the deployment process.

Continuous Maintenance Phase — Regular maintenance of both physical and virtual machines is an uncontroversial necessity to protect them against attacks and problems related to bugs. In contrast to the continuous execution model, virtual machines in the on-demand execution model are not running at all times but instead they appear and disappear. Thus, they cannot be maintained using the standard tools that are designated for physical machines that are continuously running (and thus also for virtual machines in the continuous execution model). Instead, new approaches for maintaining virtual machines even during phases of dormancy are required.

Solutions that enable continuous maintenance for virtual machines are described in Chapter 6.

Execution Phase — The need for security monitoring during the execution phase in a virtual machine's lifecycle that has already been identified in the analysis of the continuous execution model also applies to the on-demand execution model. The need for novel, virtualization-aware monitoring tools exists also for this execution model.

Furthermore, the dynamics introduced into a system by allowing on-demand provisioning force the operators of Cloud infrastructure to closely monitor the workload on all execution hosts and quickly react to overload situations. This is of particular importance in the face of Service Level Agreements (SLAs) that guarantee specific properties of the virtual infrastructure to customers.

One of the most commonly used approaches to manage the workload of the execution hosts is the migration of virtual machines to other execution hosts with a lower workload. Since the migration process of a virtual machine contains a period of time during which the virtual machine is suspended and thus not reachable, it needs to perform the migration very quickly. This is of particular importance if the migration of a virtual disk is part of this process.

A concept for virtualization-aware security monitoring of virtualized environments is presented in Chapter 6 and a solution that enables efficient migration of virtual machines including virtual disks is described in Chapter 7.

3.4 Summary

This chapter has examined the lifecycle of virtual machines in the following usage models:

- Continuous Execution Model
- On-demand Execution Model

The continuous execution model is mostly used in scenarios that deal with consolidation of physical machines. Virtual machines behave very much like physical machines in this usage model, thus existing tools can be reused to manage them. The continuous execution model is not the prevalent usage model in Virtualized Grid or Cloud Computing, and thus not the focus of this thesis.

The on-demand execution model, the prevalent usage model in both Virtualized Grid and Cloud Computing, has a great impact on the management of virtual machines. The dynamic nature of this model is challenging for both the storage system, to be able to store a large number of dormant virtual machines in a suitable way, and the physical infrastructure as a whole, to allow efficient deployment and fast migration of virtual machines. Furthermore, existing tools cannot be used to maintain dormant virtual machines, increasing the risk of running virtual machines with outdated software. This model is the focus of this thesis. The thesis proposes seven novel approaches to deal with the requirements of virtual machines in the on-demand execution model.

This page is intentionally left blank.

"Good composition is like a suspension bridge - each line adds strength and takes none away."

Robert Henri (1865–1929)

4 Virtual Machine Image Composition

4.1 Introduction

In the prevalent usage model of virtual machines in Virtualized Grid and Cloud Computing and the corresponding virtual machine lifecycle, as described in Section 3.3.2, the deployment of a virtual machine is a very frequent operation. Furthermore, the time required to provision a virtual machine is dominated by the time need for deploying the corresponding virtual machine image to the selected execution host. Optimizations of the deployment process thus improve the elasticity of the overall system. Especially for Cloud Computing, elasticity is a key performance indicator directly perceivable by customers and one of the main reasons to use Cloud infrastructures in the first place.

Depending on the architecture of the storage system for virtual machines, the deployment process might involve copying of disk images to one or more execution hosts. This time-consuming task can be bypassed if the virtual machine images are accessible via a network file system. When a suitable network file system is not available, the disk image of a virtual machine needs to be copied to a local hard disk of the execution host before the virtual machine can be started. Both approaches have pros and cons that are beyond the scope of this thesis, and use cases for both approaches exist. The remainder of this chapter focuses on use cases without a suitable network storage for virtual machine images. Therefore, copying of virtual machine images to execution hosts is required, and this accounts for a majority of the deployment time. Any technique that reduces the time required to copy disk images will thus optimize the deployment phase as well.

There are three obvious approaches that can be used to reduce the time required to copy disk images to an execution host:

1. Usage of efficient transfer mechanisms for distributing virtual machine images to execution hosts.
2. Reduction of the size of virtual machine images.
3. Caching of virtual machine images at execution hosts.

The first two approaches directly reduce the time required to copy the disk images to the execution host. The last approach, on the other hand, tries to facilitate reuse of images that have already be copied to the execution host during an earlier deployment phase. The caching approach sounds promising, because it can supersede the copy process altogether in some cases, but it introduces new two new problems: keeping the cached images in a pristine state so they can be reused and providing enough space for a number of potentially huge virtual machine images. The virtual machine image composition proposed in this chapter is aimed at reducing the size of virtual machine image. A side benefit of the image composition technique is the elimination of the caching-related problems described above.

Traditionally, virtual machines in Virtualized Grid and Cloud Computing environments are based on self-contained disk images stored as files, each disk image representing a virtual hard disk or a virtual partition in the case of Xen PV virtual machines. There are parts of virtual machine images that are likely shared by many virtual machines, e.g., the operating system or commonly used shared libraries. Since these parts are stored repeatedly in each virtual machine image, the shared parts have to be transferred to the execution host over and over again, whenever a virtual machine is deployed. Taking advantage of the similarities between different virtual machine images can not only prevent continuous retransmission of the same parts of virtual machine disk images, but also reduces the storage requirements for the virtual machines. Especially with respect to the frequency of virtual machine deployment operations, such an approach seems promising.

In this chapter, a novel approach for providing disk images for Linux-based virtual machines is presented: the *Marvin Image Compositor* (also referred to as Image Compositor). It is based on the idea of composition: a disk image containing a base installation of an operating system is dynamically combined with another disk image containing the actual application yielding a *composite disk image*. The composition is non-invasive, i.e., the disk images – also called *layers* – involved are not modified and can thus be reused in other compositions.

The reusability of disk images makes the creation of an *image cache* at the execution host feasible. This cache facilitates keeping a set of commonly required layers at every host and to have them immediately available. Even if a virtual machine based on a composite disk image has never been deployed on a specified execution host, at least one of the layers building the composite disk image is likely stored in the image cache: the layer containing the operating system. If this virtual machine is to be deployed to this execution host, only the (usually smaller) disk image containing the application needs to be transferred. This does not only improve the deployment time, but also reduces the load on the network and thus alleviates the problem of virtual machine distribution in Virtualized Grid and Cloud Computing.

Furthermore, the size of the virtual machine images kept in the storage system can be significantly reduced by this approach. Virtual machine image composition effectively applies high-level deduplication by storing only once those parts of a virtual machine image that are likely shared by many virtual machines, e.g., the operating system and commonly used shared libraries.

While the primary focus of virtual machine image composition is the improvement of deployment efficiency, the use of composite disk images also affects the creation, storage, and execution phases in the lifecycle of a virtual machine in different ways. Since the composition is done at runtime, the execution phase is obviously affected by the use of composite disk images. Storage efficiency is optimized by the use of a shared base layer that the creation of a new virtual machine can build upon. The phases in the lifecycle of a virtual machine that are related to image composition are shown in Figure 4.1.

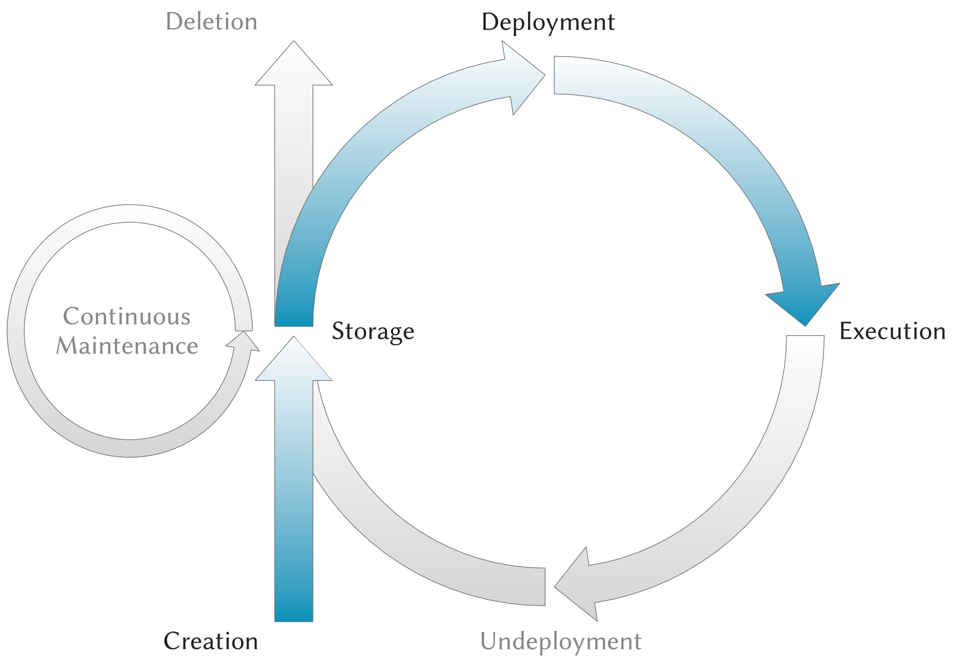


Figure 4.1 Related Virtual Machine Lifecycle Phases. The primary focus of virtual machine image composition regarding the lifecycle of a virtual machine is the deployment phase. The creation, storage and execution phases are affected as well.

Parts of this chapter have been published in [141, 142, 43].

4.2 Related Work

4.2.1 Distribution of Virtual Machine Images

We have previously worked on different virtual machine distribution methods to improve the deployment time of virtual machines in Virtual Grid and Cloud Computing environments. Thereby, we implemented a custom binary tree transfer method, a peer-to-peer approach based on the BitTorrent protocol, and a multicast transfer mechanism [43, 138]. These methods are targeted at scenarios in which a virtual machine's image file is transferred completely to an execution host before it is started.

An alternative are streaming approaches that do not transfer the entire image file to an execution host, but rather stream required chunks of the image file to the execution host on demand, i.e., when the virtual machine tries to read these chunks, and cache these chunks locally [27, 1, 127, 152]. Using these approaches, virtual machines can be started immediately at the cost of I/O performance when reading from disk if the corresponding chunk is not cached.

Nicolae et al. [106] have proposed a distributed repository for virtual machine images. The authors advocate aggregating a part of the execution hosts' local disks to build a large distributed storage system. Like in the other streaming approaches listed above, chunks are fetched on demand and cached locally to facilitate starting virtual machines immediately. A prefetching mechanism is used to fetch and cache chunks before they are actually accessed based on so-called prefetching hints that are generated while the system learns the access pattern. This improves the I/O performance especially during boot time that typically has deterministic access patterns.

Al-Kiswany et al. [6] have proposed *VMFlock*, an approach for efficient deployment of groups of virtual machine images, e.g., a multi-tier application. They use deduplication to reduce the amount of data that needs to be transferred to the execution hosts, both within the group of virtual machine images and between the group and images already available at the execution hosts. VMFlock also enables starting the virtual machines with only partial images available at execution hosts. Like the proposal of Nicolae et al., VMFlock uses previously observed access patterns to prioritize important parts when transferring the remainder of the images.

4.2.2 Reduction of the Size of Virtual Machine Image

VirtuaLinux [7] uses the *Enterprise Volume Management System* (EVMS) [42] as its storage. It relies on EVMS snapshots, i.e., persistent views of a virtual machine's disks at a specific point in time, in order to efficiently store derived versions of a virtual machine. EVMS provides no easy way to extract the differences between a snapshot and the current state of a virtual machine image, thus an efficient transfer of virtual machine images is not possible.

Parallax [166, 99] uses a custom mechanism for storing virtual machine images and creating snapshots. Template images are used to build new virtual machine 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 storage system instead of local storage.

To reach the goal of fast migration of virtual machines, Sapuntzakis et al. [132] propose a similar concept. Virtual machine 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.

A completely different approach is used in *Ventana* [116]. Instead of virtual machine images as a virtual counterpart to physical discs, views of a virtual file system 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 virtual machine called Host Manager is used to provide the view as NFS share, while the actual data is stored on on external metadata and object servers and accessed using a specialized protocol. This solution allows the reuse of common parts of a virtual machine image, but relies on fast networks to be usable.

XenoServer [80] uses *Network File System* (NFS) to access the root file system in the virtual machine, which is provided by another virtual machine called Stacking COW server running at the same host. The file system consists of a local template and one or more virtual machine 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 file system is built or on the overlays itself. While the idea is generally comparable to Image Composition, the additional virtual machine needed to provide the file system over NFS causes a performance degradation, because every I/O operation not only involves a context switch to the Virtual Machine Monitor, but also to another virtual machine 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 file system is built inside the virtual machine. Unfortunately, no details are given about this approach, except that the overlay is fetched from some network server.

A proposal for virtual machines for distributed workstations that can be used as *Condor* nodes or virtual cluster has been made by Wolinski et al. [170]. Apart from features like automatic network configuration, IP over P2P, etc. the paper proposes a stacked file system based on UnionFS. Although there are similarities to the Marvin Image Compositor, their proposal is less flexible: it is limited to two or three layers, whereby two of these layers are reserved for specific parts of the file system, it does not offer the option to include a RAM disk as layer, and lacks dynamic configuration mechanisms like preseeding and composition scripts.

Ha et al. [64] have proposed *VM overlays* to reduce the size of virtual machine images. Their aim is to improve the provisioning time of *cloudlets*, i.e., trusted small-scale Cloud data centers at the edge of the Internet that provide services to mobile devices nearby [133]. The approach uses a base virtual machine that is already available at the cloudlets and creates a VM overlay by computing the differences between the base virtual machine and the actual virtual machine. When the virtual machine is started at any cloudlet, only the differences need to be transferred. They are applied at the base virtual machine to reconstruct the actual virtual machine. VM overlays bear resemblance to layers used for image composition, but the former are not restricted to virtual hard disks, but also include memory snapshots.

Additional techniques to further reduce the storage requirements of virtual machines are described in Section 5.2.

4.2.3 Caching of Virtual Machine Images

Sotomayor et al. [150] propose virtual machine image templates for Virtualized Grid Computer. Image templates are generic reusable virtual machine images containing both an operating system and a set of tools targeted at a specific user group. At runtime, an image is created from the template and missing parts are added to that image. The template is not modified in this process and can thus be used again. Contrary to regular image files, virtual machine image templates can be cached at execution hosts.

Jeswani et al. [73] propose *DiffCache*, a technique for differential template caching to reduce the size of the template cache. Contrary to Sotomayor et al., they use the term template to denote ready-to-use virtual machine images in a cache. To facilitate their reusability, templates are copied into image files whenever an instance of a corresponding virtual machine is started. Like the Image Composition approach, DiffCache is based on the fact that different templates often have a high amount of similarities. These similarities are exploited by storing only a few base templates and a set of patches, i.e., files capturing the differences between a base template and the target template. A patch can be used to transform a base template into the target template.

A similar approach is proposed by Zhou et al. [178]. They partition the templates to be cached into clusters using *k*-means clustering. Each cluster contains a base template and enough information to reconstruct all other templates of this cluster by transforming base template.

Razavi and Kielmann [125] also advocate caching of virtual machines, but in contrast to the previous two approaches they propose caching of chunks of image files instead of entire images files to reduce the size of the image cache.

4.3 Design

Virtual machines are traditionally based on disk images comprising operating system, shared libraries, and applications. Some parts of those disk images, e.g., the operating system and some of the shared libraries, are typically shared by multiple virtual machines. Nevertheless, a copy of those shared parts is contained in every virtual machine image. These images are thus *self-contained*. A traditional virtual machine is depicted in Figure 4.2.

Virtual machine image composition is based on the idea of separating disk images into multiple *layers* that encapsulate the different parts of the disk image: a part containing the operating system and commonly used shared libraries likely shared by many virtual machines, and a part containing the virtual machine specific part, i.e., applications and their dependencies. A *composite disk image* is built from those layers at runtime, providing a complete and consistent view to users and applications. The process of building the composite disk image is called *image composition*, the tool implementing the composition is called *Image Compositor*.

Figure 4.2 Traditional Virtual Machine. A traditional virtual machine is based on a self-contained disk image comprising both common and virtual machine specific parts (depicted by the green and orange areas of the disk image, respectively). It is important to note that there is no clear distinction between those parts (depicted by the use of a gradient).

The layers can be seen as *virtual machine building blocks* that can be composed to build different virtual machine images. In the remainder of this thesis, virtual machines based on a composite disk image are called *layered virtual machines*.

Three usage scenarios for the image composition technique are shown in Figure 4.3. All disk images are based on a common *base layer* that contains a generic installation of an operating system and commonly used shared libraries. In the first scenario, a two-layer image is composed of the base layer and an *user layer* that contains software installed by a user. This scenario is the equivalent to a regular virtual machine that is created by a user without image composition. Similarly, in the second scenario, the image is composed of the base layer and a *vendor layer* that contains one or more preinstalled software products of a vendor. Providing a vendor layer with preinstalled, directly usable software eases the burden of installation and configuration on the user and is a potential selling point. In the last scenario, a three-layer image is composed of the base layer, a vendor layer and a user layer on top of it. This scenario combines the former two scenarios to build an image containing both preinstalled software as well as further extensions or software installed by the user. These are the most common scenarios, although the image composition technique does not restrict the user to any of these scenarios.

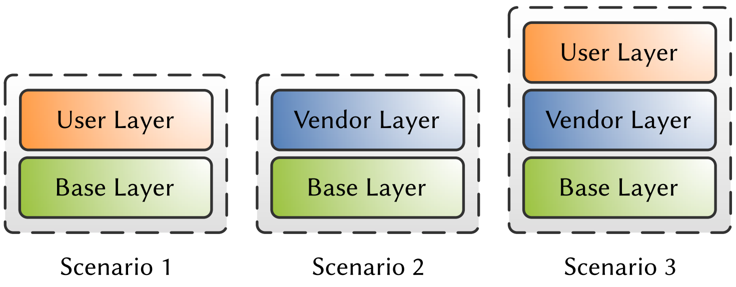


Figure 4.3 Usage Scenarios for a Layered Virtual Machine. In scenarios 1 and 2, the virtual machines are based on a composite disk image consisting of two layers: a common base layer and a user- or vendor-created layer, respectively. Scenario 3 combines the first two scenarios by including both the user- and vendor-created layer in the composite disk image.

To facilitate reuse of layers, e.g., the base layer in all 3 scenarios above or the vendor layer in the second scenario, the composition mechanism has to be non-invasive and thus guarantee that the involved layers are not modified. This can be achieved by including a *temporary layer* into the composite disk image, so that all write operations to the composite disk image are conducted on the temporary layer. The term temporary layer implies that this kind of layer is discarded after the virtual machine using the corresponding composite disk image is shut down. All non-temporary layers are called *persistent layers*, because they can and should be reused. Figure 4.4 again shows the three scenarios described above when a temporary layer is used. Such a temporary layer can be based on a temporary virtual hard disk or a RAM disk, depending on the amount of write operations that are expected during the execution of the virtual machine.

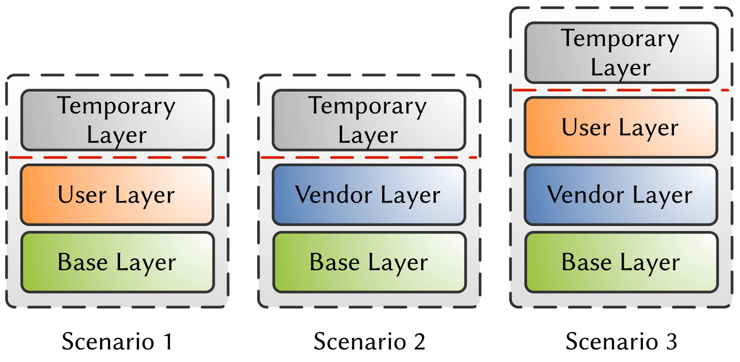


Figure 4.4 Composite Disk Image with Integrity Guarantee. This figure shows the three scenarios from Figure 4.3 with a temporary layer. Only the temporary layer is writable (the write barrier is shown as a dashed, red line), whereas all other layers are write-protected. This setup protects the involved layers from changes.

If the modification of a layer is intentional, for example, to create a new layer or to update an existing one, no temporary layer is used, so that all write operations are conducted on the highest layer in the composite disk image. Figure 4.5 again shows the three scenarios described above when no temporary layer is used.

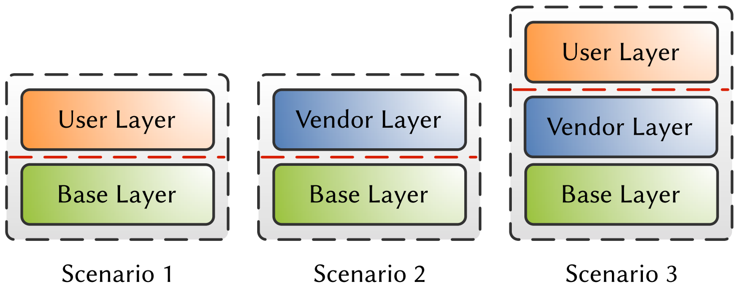


Figure 4.5 Modification of a Layer in a Composite Disk Image. This figure shows the three scenarios from Figure 4.3 without a temporary layer. Only the highest layers are writable (the write barrier is shown as a dashed, red line), whereas all other layers are write-protected. This setup facilitates the modification of the highest layers.

By decomposing the virtual machine image into layers and basing virtual machines on a shared base layer, the size of the individual layers is reduced in comparison to a regular virtual machine image. In combination with image caches at the execution host and the reusability of layers enabled by the use of temporary layers, this can reduce the deployment time of virtual machines as well as the network load during deployment, if the base layer is already available at the execution host.

Another important benefit of the image composition technique is the possibility to use a single disk image concurrently in multiple virtual machines. Without image composition, each virtual machine requires its own copy of the disk image. This applies even to cloned virtual machines (see Section 3.3.2) that are based on a single disk image. An example with two cloned virtual machines is depicted in Figure 4.6. Before the first virtual machine can be started, the disk image has to be copied into the local storage of the execution host (Figure 4.6a). Note that the image cache on the execution host is optional in this case. Without an image cache, the disk image is copied from the storage system. For a second virtual machine, another copy of the disk image is required (Figure 4.6b).

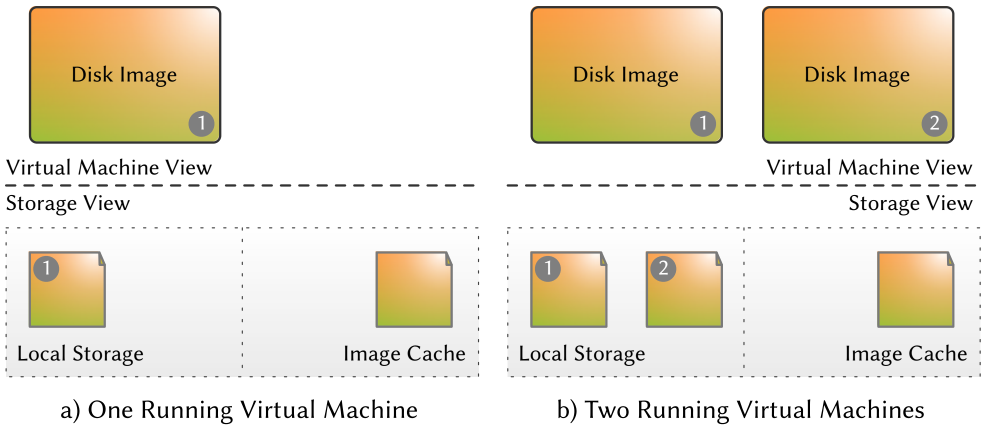


Figure 4.6 Cloning of Traditional Virtual Machines. Contents of the local storage of an execution host when two cloned virtual machines are started consecutively. Individual copies of the disk image will be created for each of the two virtual machines. Note that the image cache is optional.

The same scenario of two cloned virtual machines is depicted in Figure 4.7 for layered virtual machines. The two layers required for the composite disk image are copied to the image cache of the execution host and are used directly, i.e., no further copying of those layers is required. A temporary layer is created either in the local storage of the execution host (Figure 4.7a) or in its RAM (not depicted). For the second virtual machine, only a new temporary layer needs to be created. The other two layers are used as-is (Figure 4.7b).

The remainder of this section is structured as follows. In Section 4.3.1, a list of requirements for image composition is defined. Sections 4.3.2 and 4.3.3 describe two concepts that are essential for the design of an image composition system: the actual composition technology and the Linux boot process. Based on these concepts and the requirements, Section 4.3.4 illustrates the design of the Image Compositor.

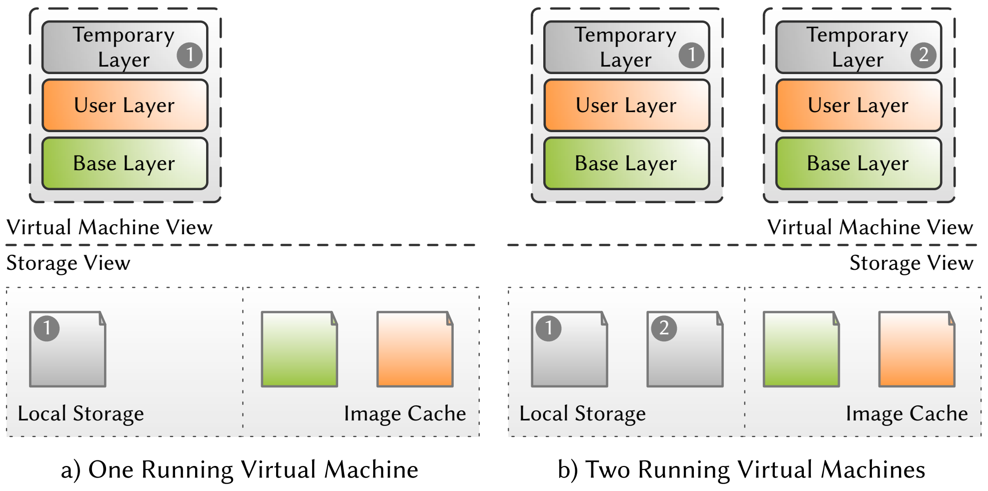


Figure 4.7 Cloning of Layered Virtual Machines. Contents of the local storage and image cache of an execution host when two cloned virtual machines are started consecutively. Both virtual machines share the layers in the image cache. No additional copying of those layers is required. Only an individual temporary layer must be created for both virtual machines.

4.3.1 Requirements

An Image Compositor needs to satisfy all requirements listed in this section to provide the desired functionality. The requirements are divided into two parts: fundamental requirements that are absolutely necessary to implement image composition and specific requirements for Virtualized Grid Computing environments.

4.3.1.1 Fundamental Requirements

The following list contains the fundamental requirements on an Image Compositor:

R4.1 Create composite disk images that are composed of arbitrary layers.

R4.2 Create composite disk images that are built from an arbitrary number of layers.

R4.3 Create composite disk images that contain read-only layers and are writable nevertheless.

R4.4 Create composite disk images that use RAM disks as temporary layer.

R4.5 Create composite disk images that are based on a configuration provided at composition time.

R4.6 Create composite disk images that are transparent to application software.

Requirement R4.1 ensures the flexibility of the Image Compositor that is required to support the usage scenarios for layered virtual machines defined above (see Figure 4.3).

While there are obviously dependencies between layers, i.e., a composite disk image built from a base layer and a user layer defines a working disk image, while a composite disk image built only from a vendor layer and a user layer does not, there should be no restrictions whatsoever imposed by the Image Compositor that prevent composing arbitrary layers. Therefore, a user or vendor layer can be combined with a base layer other than the one it was build with.

That flexibility comes at the price of potentially introducing compatibility problems between layers that are combined in a composite disk images. For example, if the base layer is replaced with another layer that does not contain all the packages available in the original base layer, software installed in the upper layer might not work anymore. Thus, logical compatibility of layers has to be guaranteed by the virtual machine management system that has to store the composition of layers for each virtual machine. A compatibility checker for different layers is presented in Section 6.3.2. It is intended to ensure the compatibility of a user or vendor layer with an updated base layer, but can also be used to verify layer compatibility in general.

Since the usage scenarios for layered virtual machines defined in Figure 4.3 are only examples, other use cases with more complex compositions of layers are possible and should be supported. Requirement R4.2 ensures that the Image Compositor is not limited to fixed compositions of layers like the ones defined in the scenarios and supports other use cases as well.

Reusability of layers is the basis for reducing the deployment time of virtual machines. Since reusability is only guaranteed if the layers are not changed, they have to be included in the composite disk image in a read-only fashion. Requirement R4.3 ensures that the Image Compositor supports this mode of usage and provides the necessary means to redirect write operations to a temporary layer.

In some use cases, the creation of a virtual hard disk based temporary layer is either not necessary or not possible. Examples of such use cases are virtual machines running software that barely writes to disk or execution hosts without local storage for the virtual hard disk. For those cases, Requirement R4.4 ensures that the Image Compositor supports the use of RAM disks as temporary layer.

Requirement R4.5 ensures that the flexibility provided by Requirements R4.1 to R4.4 can be utilized. Instead of a fixed layer composition that is defined at the creation time of the virtual machine and stored in the highest layer, each time the virtual machine is started a specific composition is selected and the corresponding configuration is passed to the Image Compositor that combines the layers accordingly. This facilitates the recombination of layers to build new composite disk images without modifying the involved layers. Such a modification does not only violate Requirement R4.3, but also compromises the reusability of the layer. Furthermore, this dynamic configuration at composition time also is the key enabler of switching between the read-only composition (see Figure 4.4) that protects all involved layers from changes and the writable composition (see Figure 4.5) that enables modifying the highest layer.

Finally, Requirement R4.6 ensures that the use of a composite disk image as virtual machine image is transparent to application software running inside the virtual machine. The composition may not affect or break application software in any way. This transparency requirement does not include system software, i.e., the operating system or software for system maintenance that may or even has to be able to detect the presence of a composite disk image depending on the implementation technique used.

4.3.1.2 Grid-specific Requirements

As Foster stated in his three point checklist [51], the first characteristic of a Grid is that it coordinates resources from different Grid sites, i.e., different administrative domains without centralized control. To put it simply, Grid Computing is about executing jobs spanning either a single or multiple Grid sites. In Virtualized Grid Computing, jobs are executed inside virtual machines that are prepared by the user, typically at his local Grid site. Applying the first characteristic of Grid Computing to Virtualized Grid Computing implicates that these virtual machines have to be exchanged between and executed on different Grid sites.

Although virtual machines are self-contained, some dependencies to the execution environment remain. Such dependencies include, but are not restricted to: network infrastructure, Grid head node and job scheduler, as well as home directories that are usually provided via Network File System (NFS) [146] or comparable distributed file systems. These services are provided by the Grid site and are very likely heterogeneous because of the architecture of the Grid. A specific configuration of the virtual machine is thus required to be able to execute it on a Grid site other than the one it was built for. While the network configuration can be automated via the Dynamic Host Configuration Protocol (DHCP) [39], the configuration of the other required services has to be done manually inside the virtual machine to make it fully usable. Unfortunately, the Grid site that is used to execute a job, and therefore the specific execution environment, is not known in advance, but selected by the scheduler based on the workload of the different sites and other parameters. Thus, configuration information for *all* Grid sites has to be embedded into the virtual machines together with a tool that automatically selects and applies the right configuration at boot time.

Another way to solve this configuration problem is to employ image composition and use a *site layer* that contains the specific configuration of a Grid site. The site layer is added to the composite disk image at boot time. While this is a feasible approach, it increases the complexity of the composite disk image by adding a complete layer, when actually only a few configuration files have to be added or replaced. It is more efficient to do the necessary configuration changes in the virtual machine without adding another layer to the composite disk image.

This leads to the following two specific requirements that have to be satisfied by an Image Compositor that is applicable in Virtualized Grid Computing:

R4.7 Create composite disk images that contain additional files provided externally via an injection mechanism at boot time.

R4.8 Create composite disk images that can be modified by scripts, both stored inside the layers and injected, during the composition of the layers.

Requirement R4.7 ensures that files in a composite disk image can be easily replaced or new files can be added without adding a complete layer to the composite disk image. Files from an external source are injected into the composite disk image by *preseeding* the temporary layer, i.e., copying the files to the temporary layer before the composite disk image is build. Preseeding should be limited to temporary layers, to keep the persistent layers in a pristine state and ensure their reusability. This mechanism is intended for adding configuration files to a virtual machine to adopt it to a new execution environment, so this restriction does not limit the usability of the overall system.

In some cases, adding or replacing files might not be enough to adopt the configuration of a virtual machine to the execution environment. Arbitrary configuration changes can be done using scripts that are executed inside the virtual machine at composition time. Requirement R4.8 ensures that the Image Compositor provides an option to execute such *composition scripts*. The scripts can either be provided as part of one of the layers that is used to build the composite disk image or injected using the preseeding mechanism described above.

4.3.2 Composition Techniques

The actual composition of layers can be achieved at two different levels: at the level of block devices and at the level of file systems. The former approach implies that the Image Compositor has to provide a virtual block device containing a single, combined file system with the contents of all layers. The latter approach implies that the Image Compositor uses multiple, independent block devices containing individual file systems that are combined at the virtual file system (VFS) abstraction layer in the kernel. In this section, both approaches are compared and their advantages and disadvantages are assessed in the context of virtual machine image composition.

At the block device level, a promising technology to implement the composition of layers is redirect-on-write. It is widely used in the snapshot feature of modern virtualization products that enables users to roll back any changes to a virtual machine by returning to an older state of a virtual hard disk. The latest version of *Linux Logical Volume Manager (LVM)* [98] also enables the creation of writable snapshots that are based on redirect-on-write. At the file system level, union mounts are the suitable technology to implement composition of layers. For Linux, the most prominent implementations are *aufs* [108] and *UnionFS* [173, 122, 157]. Other implementations exist, but they are less widely used. Support for one or both of these two tools is available in almost all Linux distributions.

4.3.2.1 Block Device Level: Redirect-on-write

Redirect-on-write technology applied at the block device level (referred to as block-level in the following) is illustrated in Figure 4.8. In this example, a composite disk image consisting of a read-only layer that contains the two files `file1` and `file2` in the directories `dirA` and `dirB`, respectively, and a writable layer. Initially, the writable layer is empty (Figure 4.8a). When `file1` is modified, the new blocks are written –

or redirected – to the writable layer (Figure 4.8b). While the unmodified blocks of `file1` are accessible through the writable layer, the modified blocks are hidden by the blocks in the writable layer. Adding the new file `file3` to `dirA` is easier: all its blocks are written to the writable layer (Figure 4.8c). They hide the empty blocks in the read-only layer.

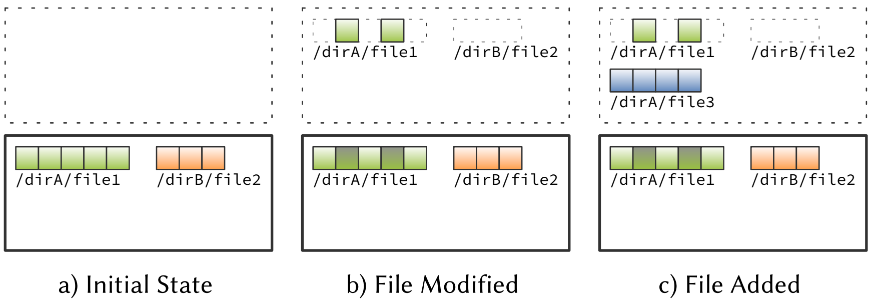


Figure 4.8 Redirect-on-write Example. A composite disk image consisting of a read-only layer and a writable layer, combined with block-level redirect-on-write technology. The contents of the layers are shown in the initial state a), after `file1` is modified b), and after `file3` is added c). The outlines of both `file1` and `file2` are illustrated in the writable layer to show the 1:1 correlation between block numbers in both layers.

There is a 1:1 correlation between block numbers in both layers that are combined with block-level redirect-on-write technology. For instance, to modify block `1234` of the read-only layer, the modified block needs to be stored at block `1234` of the writable layer[1]. Thus, this technology induces a tight coupling between the layers of a composite disk image. Therefore, the underlying layer in a composite disk image implemented using this technology must not be changed, because those changes will very likely result in broken file system structures or worse errors.

The redirect-on-write approach is related to the copy-on-write approach that was used for read-only snapshots in the first version of LVM. Often, the term copy-on-write is synonymously used for both technologies, although their goals are completely different. Regarding the example in Figure 4.8, redirect-on-write ensures that the underlying layer is kept in pristine state and all write accesses are redirected to the upper layer. On the other hand, copy-on-write would copy the unmodified blocks from the underlying to the upper layer before writing modified blocks to the underlying layer. The fundamental difference is that after throwing away the upper layer, the underlying layer would contain the pristine state using redirect-on-write and the state after the last write access using copy-on-write. Thus, the copy-on-write technology cannot be used to implement an Image Compositor without breaking Requirement R4.3.

An assessment of block-level redirect-on-write technology regarding the requirements of composite disk images is given in Table 4.1.

[1] Most implementations of redirect-on-write approaches use some kind of mapping to enable arbitrary placement of blocks. Thus, the writable layer, or to be precise the underlying file, can be much smaller than the read-only layer it has been combined with, depending on the number of changes.

Table 4.1 Assessment of Redirect-on-write Technology. An assessment of the suitability of block-level redirect-on-write technology for providing composite disk images.

Requirement		Assessment of Redirect-on-write Technology
Fundamental Requirements		
R4.1	×	Because of the tight coupling between layers it is not possible to combine arbitrary layers to a composite disk image.
R4.2	✓	There is no layer limit for the number of layers that can be combined to a composite disk image using the block-level redirect-on-write technology.
R4.3	✓	This requirement is satisfied when the redirect-on-write technology is used together with a writable temporary layer.
R4.4	✓	The redirect-on-write approach is independent on the storage location of the writable layer.
R4.5	✓	Because the composition of the disk image is done outside the virtual machine (see R4.6), configuration at composition time is not an issue.
R4.6	✓	A composite disk image built using block-level redirect-on-write technology is exposed as a file or block device that can be used as disk image for a virtual machine. The composition is done outside the virtual machine and thus completely transparent to both application and system software.
Grid-specific Requirements		
R4.7	✓	Because the composition of the disk image is done outside the virtual machine (see R4.6), additional files can be easily added to the composite disk image during the composition process.
R4.8	(✓)	Because the composition of the disk image is done outside the virtual machine (see R4.6), the execution of scripts during the composition is a potential security risk, at least for scripts included in the layers, because they might be malicious.

✓: requirement satisfied, ×: requirement not satisfied, (✓) : requirement partly satisfied

4.3.2.2 File System Level: Union Mounts

The union mount technology applied at the file system level is illustrated in Figure 4.9. The example corresponds to the block-level redirect-on-write example in Figure 4.8. Again, there is a read-only layer containing two files and an initially empty, writable layer (Figure 4.9a), except that in this case a file system level view is given. When `file1` is modified, it is first copied to the writable layer. After a copy of `file1` exists in the writable layer, all read and write accesses to `file1` are redirected to the writable layer. The old version of `file1` in the read-only layer is hidden by the new version in the writable layer (Figure 4.9b). The newly added file `file3` is directly written to the writable layer (Figure 4.9c).

To emphasize the differences between the redirect-on-write approach at the block-level and union mounts at the file system level, a block-level view of the layers from

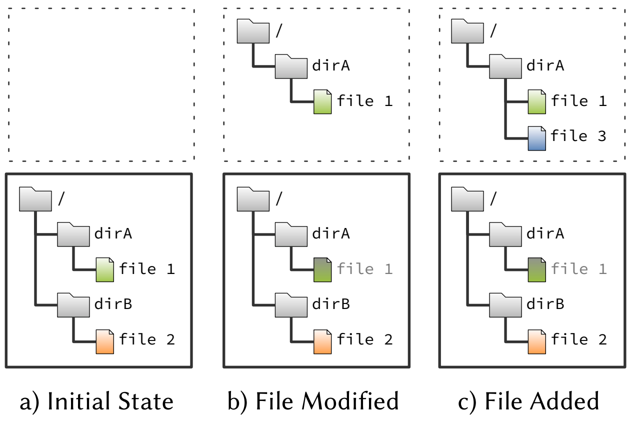


Figure 4.9 Union Mount Example (File System Level View). A composite disk image consisting of a read-only layer and a writable layer, combined with a union mount. The contents of the layers are shown in the initial state a), after `file1` is modified b), and after `file3` is added c).

Figure 4.9 is given in Figure 4.10. When `file1` is modified (Figure 4.10b), the complete file is copied to the writable layer. Then the modifications are written to the copy, depicted by the emphasized blocks 2 and 4 of the file in the writable layer. In contrast to the block-level redirect-on-write approach, the unmodified blocks are part of the writable layer too. The original file in the read-only layer is hidden at the file system level, not at the block-level. This fact is emphasized by Figure 4.10c, where `file3` apparently overlays `file2`, without actually hiding it. Thus, the coupling between the two layers is loose.

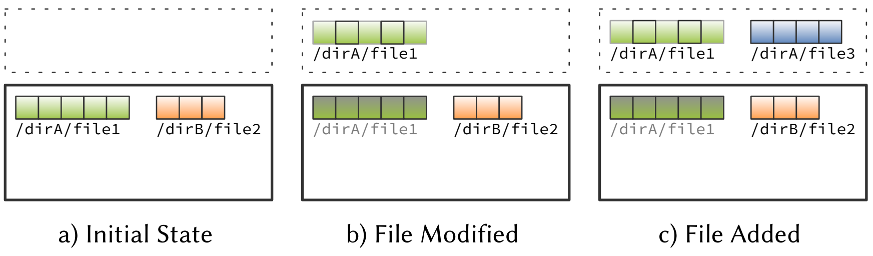


Figure 4.10 Union Mount Example (Block-level View). The corresponding block-level view of the composite disk image that is combined with a union mount (see Figure 4.9). The contents of the layers are shown in the initial state a), after `file1` is modified b), and after `file3` is added c).

In contrast to the block-level approach, the result of composition using the file system level approach is not exposed as a disk image, but as file system. Unfortunately, a file system cannot be used as virtual hard disk for a virtual machine, because virtual hard disks need to be either a block device or an image file. Thus, the composition has to be done inside the virtual machine. The file system level approach is thus a bit less transparent compared to the block-level approach. Technically, both the block-level and the file system level approach are redirect-on-write technologies, although union mounts rely on a copy-up process that copies files to be modified completely to the writable layer before the modified data is written.

An assessment of union mounts regarding the requirements of composite disk images is given in Table 4.1.

Table 4.2 Assessment of Union Mounts. An assessment of the suitability of union mounts for providing composite disk images.

Requirement		Assessment of Union Mounts
Fundamental Requirements		
R4.1	✓	Because of the loose coupling between layers arbitrary layers can be combined to a composite disk image.
R4.2	✓	There is no layer limit for the number of layers that can be combined in a composite disk image using a union mount.
R4.3	✓	This requirement is satisfied when the union mount includes a writable temporary layer.
R4.4	✓	The union mount approach is independent of the storage location of the writable layer.
R4.5	(✓)	Because the composition of the disk image is done inside the virtual machine (see R4.6), the configuration has to be passed into the virtual machine to the Image Compositor. Different options to solve this problem are described in Section 4.3.4.
R4.6	(✓)	A composite disk image built using union mount technology is exposed as file system and cannot be used as disk image for a virtual machine. Thus, the composition has to be done inside the virtual machine. This kind of composition is visible to system software, but typically transparent to application software.
Grid-specific Requirements		
R4.7	(✓)	Because the composition of the disk image is done inside the virtual machine (see R4.6), additional files have to be passed into the virtual machine to the Image Compositor. Different options to solve this problem are described in Section 4.3.4.
R4.8	✓	Because the composition of the disk image is done inside the virtual machine (see R4.6), the execution of scripts during the composition is secure even for potentially malicious scripts included in the layers, because their impact is limited to the virtual machine.

✓ : requirement satisfied, (✓) : requirement partly satisfied

4.3.2.3 Discussion

A summary of the suitability assessments of the redirect-on-write and union mount approaches for implementing composite disk images is given in Table 4.3 by summing up the satisfied, partly satisfied and unsatisfied requirements. Since neither of the approaches does completely satisfy all requirements, the partly satisfied and unsatisfied requirements are examined below before a decision between the approaches is taken.

Union mounts do not completely satisfy Requirement R4.6, because the composition has to be done inside the virtual machine. The existence of the layers is thus visible to

Table 4.3 Summary: Assessment of Composition Techniques. Comparison of the numbers of satisfied, partly satisfied and non-satisfied requirements for each of the two composition techniques.

	Satisfied		
Approach	Fully	Partly	Not at All
Redirect-on-write	6	1	1
Union Mounts	5	3	0

the software inside the virtual machine, at least to the operating system, but potentially also to application software. Details about these violations of Requirement R4.6 as well as about limiting the visibility of and preventing the access to the raw layers are given in Section 4.3.4. Additionally, union mounts do not completely satisfy Requirements R4.5 and R4.7, because depending on the virtualization technology used, there is no direct way to hand over the configuration of the composite disk image or additional files for injection into the composite disk image to the Image Compositor running inside the virtual machine. For these cases, solutions will also be presented in Section 4.3.4 that help to fully satisfy the stated requirements with this approach.

The redirect-on-write approach does not completely satisfy Requirement R4.8 because of a security concern regarding the execution of potentially malicious scripts outside of the virtual machine during the composition process. It is likely possible to work around this issue by splitting the composition process into a composition phase, running outside the virtual machine, and a post-composition phase, running inside the virtual machine. In this case, the scripts are executed in the post-composition phase to limit their access to the virtual machine. On the contrary, Requirement R4.1 is not satisfied at all by the redirect-on-write approach and there is no way to work around that limitation.

The detailed examination of partially satisfied and unsatisfied requirements shows that union mounts are the appropriate technology for the Image Compositor. For all partially satisfied requirements workarounds are available that constitute acceptable solutions to satisfy the requirements of image composition. The redirect-on-write approach on the other hand fails to satisfy Requirement R4.1, one of the primary requirements to make image composition flexible enough to support all of the desired scenarios. Thus, the union mount approach is selected for the implementation of the Image Compositor.

4.3.3 Integration into the Boot Process

With the decision in favor of union mounts instead of redirect-on-write technology, the Image Compositor has to be integrated into the Linux boot process. This integration is necessary, because the image composition has to be done inside the virtual machine, as a union mount cannot be used as virtual hard disk. Finding the right position in the boot process is important to satisfy all requirements regarding image composition. In this section, an overview of the Linux boot process is given that highlights the influence of different virtualization techniques, and potential positions in the boot process are determined and assessed.

4.3.3.1 Linux Boot Process on Physical Machines

Based on the description of Jones [75], a dissection of the Linux boot process into six phases is presented in Figure 4.11. The first two phases are generic and can boot any operating system, while the last four phases are Linux specific. The individual phases are detailed below.

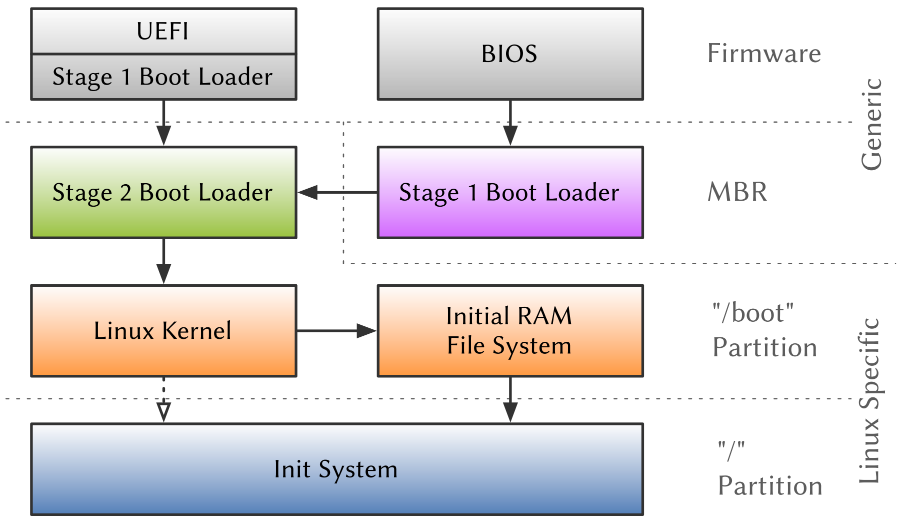


Figure 4.11 Boot Process of Linux. There are six phases in the boot process of Linux, each controlled by another component. The first two components are generic and can boot any operating system, while the latter four phases are specific to Linux.

Phase 1 - Firmware — This phase of the boot process is controlled by the firmware, a very much hardware dependent piece of software embedded into every computer system. On x86-based computers the *Basic Input/Output System* (BIOS) is the prevalent firmware interface, with its competitor *Unified Extensible Firmware Interface* (UEFI) – the successor of the *Extensible Firmware Interface* (EFI)– rapidly catching up in newer systems. UEFI aims to overcome limitations of the BIOS that date back to the first IBM PC.

The main tasks of both BIOS and UEFI are the detection and initialization of devices, e.g., CPU, RAM, video card, and disk drives. After the hardware is ready, control is transferred to the stage 1 boot loader.

Phase 2 - Stage 1 Boot Loader — This phase of the boot process is controlled by the stage 1 boot loader. In the BIOS case, it is loaded by the firmware from the Master Boot Record in the first sector of the boot disk. It consists of the actual boot loader and the partition table. The boot loader scans the partition table for the active partition, loads the stage 2 boot loader from the Volume Boot Record of that partition and transfers control to it.

In the UEFI case, the stage 1 boot loader is integrated into the firmware. It is able to read the EFI System Partition, a special partition on the boot disk containing among others the stage 2 boot loaders of all installed operating systems. If multiple operating systems are installed, the EFI Boot Manager selects the boot loader to use either by consulting its configuration or by letting the user choose the boot loader from a menu. Then, it loads the selected stage 2 boot loader and transfers control to it.

Phase 3 - Stage 2 Boot Loader — This phase of the boot process – the first that is specific for Linux – is controlled by the stage 2 boot loader. It is responsible for loading the Linux kernel and the *initial RAM file system (initramfs)*, an archive of files required for booting the system (see Linux Kernel). Both the kernel and the initramfs are regular files in either the `/boot` partition. After loading both files, the memory address of the initramfs and the kernel command line – containing additional parameters for the kernel, e.g., the root device, the mount option for the root device, and whether to boot into single user mode or not – are passed to the kernel through the kernel header structure and the kernel is started.

The most prevalent boot loader for Linux nowadays is the *Grand Unified Bootloader* (GRUB) [63]. It typically uses a configuration file on the `/boot` partition to store the parameters mentioned above, but also facilitates interactive usage for troubleshooting.

Phase 4 - Linux Kernel — In this phase of the boot process, the kernel takes over control of the system. The Linux kernel is typically stored as a compressed image, so it first decompresses itself. Then, it initializes all important subsystems and extracts the contents of the initramfs into the *rootfs*, an in-memory root file system that is always present in Linux 2.6 and later systems [82]. It is used as temporary root file system and enables the kernel to finish booting without mounting any physical device [75].

At this point, the actual root file system is not yet mounted, although the kernel has finished booting. Most likely, the kernel does not even include support for the root file system or its underlying block device, but relies on additional kernel modules. The kernel transfers the control of the boot process to the initramfs, more precisely to the *init script* inside the initramfs.

Before the initramfs was introduced, the Linux kernel used to mount the root file system itself. This functionality is still included in modern Linux kernels, although most distributions switched to using initramfs because of the improved flexibility and modularization of the kernel. This way of booting requires that all necessary drivers as well as support for the root file system is compiled into the kernel, as no modules are available before the root file system is mounted. The root file system is specified using the kernel command line. After initialization, the kernel mounts the root file system and transfers control directly to the init system. This is depicted in Figure 4.11 with the dotted arrow.

Phase 5 - Initial RAM File System — This phase of the boot process is controlled by the init script contained in the initial RAM file system that is responsible for mounting the actual root file system. It determines the root file system based on the kernel command line and mounts it. If the kernel has no built-in support for the root file system, the init script loads the required kernel modules from the initramfs beforehand.

The last and most important step in the init script is to switch the system's root directory from the initramfs to the actual root file system (*chroot*) and to transfer control to the actual init system. This is done by replacing the shell process executing the init script with the main binary of the init system (*execve*). Before this last step, it moves the special file systems `/proc`, `/dev`, and `/sys` into the actual root file system and deletes all files from the initramfs to free the memory occupied by them.

Phase 6 - Init System — The final phase in the boot process is controlled by the init system that takes care of completing the system startup by starting the required services. Depending on the Linux distribution, either a System V style init system, e.g., in Debian, *Upstart*, e.g., in Ubuntu, or *systemd*, e.g., in Fedora or openSUSE, can be used. The following description focuses on a System V style init system.

The main purpose of the init system is to launch all configured services during the system boot process and to stop them during the system shutdown. This is typically realized using a set of so-called *init scripts* that contain instructions on how to launch and deactivate a service and dependency information allowing to derive the service launch order. Advanced implementations provide additional features like service monitoring and automated restarts or crash reports.

4.3.3.2 Linux Boot Process in Virtual Machines

The description of the Linux boot process above focuses on physical machines. Depending on the type of virtualization used, the boot process of a virtual machine differs slightly from the one described above.

Full Virtualization

The full virtualization technology emulates a complete computer system to support running unmodified operating systems in virtual machines. The emulation includes not only the typical devices that the user of this virtualization technology deals with, i.e., hard disks and network adapters, but also low-level devices like the system chipset and a virtual firmware [160]. Because of this comprehensive emulation of hardware, the boot process of a virtual machine running Linux is identical to that of a physical machine. Both stages of the boot loader, the Linux kernel and the initramfs are part of the virtual hard disk of the virtual machine, so the virtual firmware can load and execute the stage 1 boot loader that continues the boot process as described above.

There is an exception to the statement above. Although KVM and Xen HVM are based on hardware-assisted full virtualization, and thus typically use the same boot process as a physical machine, both support the *direct kernel boot* mode. In this mode, which is compatible to the default boot process for Xen PV described below, the kernel, initramfs, and kernel command line can be manually specified when starting a virtual machine. The hypervisor then loads both the kernel and initramfs directly without relying on a boot loader. Nevertheless, the virtual hardware needs to be initialized, so the virtual firmware is still required. The boot process in this case thus starts with Phase 1, then the hypervisor loads the kernel and initramfs directly into the virtual machine's memory (replacing Phases 2 and 3), and finally the boot process continues as usual with Phase 4.

Paravirtualization

The paravirtualization technology can only run modified operating systems in virtual machines. On the other hand, there is no need to emulate a complete computer system, because a modified operating system does not rely on emulated hardware. Consequently, there is neither a firmware nor a boot loader that can load the Linux kernel. Since Xen is the prevalent hypervisor based on paravirtualization, this description is focusing on Xen PV virtual machines.

Traditionally, neither the Linux kernel nor the initramfs are part of a Xen virtual machine image, but instead both are kept on the file system of the dom0. Both the kernel and initramfs of a virtual machine are set in the virtual machine configuration file, together with the kernel command line. The first three phases of the Linux boot process described above do not exist in the case of paravirtualization. Instead, the hypervisor itself loads the chosen Linux kernel, initramfs and kernel command line directly into the memory of the virtual machine (replacing the Phases 1 to 3). Afterwards, it passes the memory address of the initramfs and kernel command line to the kernel through the corresponding header structures and starts the kernel. The boot process then continues as usual with Phase 4. The complete Linux boot process of a Xen virtual machine is depicted in Figure 4.12.

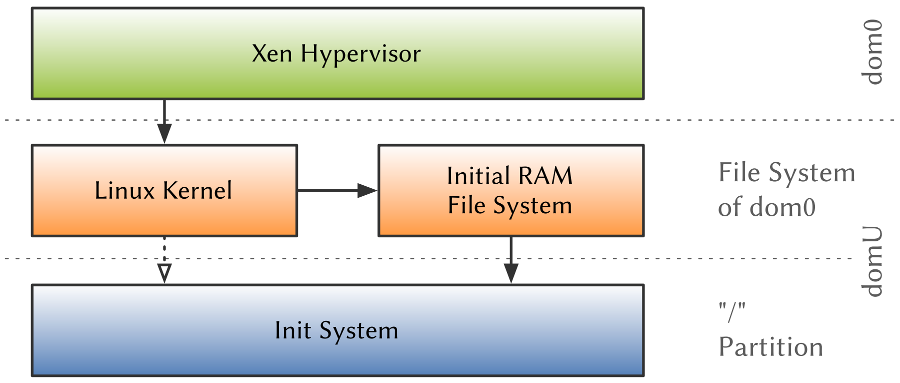


Figure 4.12 Boot Process of Linux in a Xen Virtual PV Machine. There are four phases in the boot process of Linux in a Xen virtual machine. The first component is the hypervisor that replaces the firmware and boot loaders used by physical machines and virtual machines using full virtualization. The remaining three phases are identical to the boot process as shown in Figure 4.11, except that both the Linux kernel and the initial RAM disk are provided externally.

More recently, two tools where developed that allow to store the Linux kernel, the kernel command line and the initramfs inside the virtual machine image: *pyGrub* [175] and *PV-GRUB* [176]. The former is a python script that parses a GRUB configuration file stored in the virtual machine image, extracts both the chosen Linux kernel and initramfs from the virtual machine image and copies them to a temporary directory in the file system of the dom0. From that point on, the actual boot process is identical to the traditional process described above. The pyGrub approach enables users to add additional options to the kernel command line via the virtual machine configuration file.

PV-GRUB, on the other hand, is a version of GRUB based on Xen Mini-OS [118]. Instead of firmware routines, it uses functions of Mini-OS for hardware access, e.g., reading the Linux kernel from the virtual hard disk. This approach is considered as more secure compared to pyGrub, since the parsing of the GRUB configuration is done inside the domU and noting is copied from the domU to the dom0. The hypervisor loads the PV-GRUB binary, consisting of both Mini-OS and PV-GRUB, into the memory of the virtual machine and starts it like a normal Linux kernel (replacing Phases 1 and 2). Then boot process then continues with Phase 3.

4.3.3.3 Potential Boot Process Phases for Integration

In this section, the different phases of the boot process are assessed with regard to hosting the image composition process. Of the six phases of the boot process identified in Section 4.3.3.1, the first three phases are specific to the boot process of either physical machines or virtual machines using hardware-assisted or full virtualization. The same holds true for the first phase identified in Section 4.3.3.2 that is specific to virtual machines using paravirtualization. Thus, all of these phases can be excluded from further assessment. The latter three phases, which are identical in both cases, are further assessed below.

The Image Compositor can be placed in any of the three remaining phases of the Linux boot process. Depending on the phase, different techniques have to be used to implement the Image Compositor and there are specific obstacles to overcome to make image composition working. The implementation techniques and obstacles are discussed below.

Linux Kernel — To be integrated directly into the Linux kernel, the Image Compositor has to be implemented as a Linux kernel module. This approach imposes the same restriction as the boot process without an initramfs: everything needed for the image composition, e.g., all required drivers and file system implementations, needs to be compiled into the kernel, otherwise the system cannot boot from the composite disk image.

The basic image composition functionality can be easily implemented in the kernel. On the other hand, complex parts of the Image Compositor that are needed to satisfy Requirements R4.5, R4.7 and R4.8, might be more cumbersome to implement in the kernel space. Some advanced features are even impossible to implement, because the kernel does not contain the required functionality.

One example is the injection of tar archives that are automatically extracted in the composite disk image, because the kernel cannot read tar archives. A workaround to support archives nevertheless is to use the less widely known cpio archive format that is included in the kernel because it is used to store the contents of the initramfs. Requirement R4.8 cannot be implemented at all using this approach, unless a scripting engine is added to the kernel, which is very likely a bad idea from a security standpoint.

So, the only way to satisfy all requirements is to rely on applications that are contained in the base layer of the composite disk image. This approach does not only facilitate the implementation of features like the injection of tar archives, but it is also the only reasonable way to implement the scripting support for modifications of the composite disk image during the composition (Requirement R4.8). Unfortunately, this approach imposes some requirements on the base layer used for the image composition, i.e., specific tools need to be installed in specific locations, and thus contradicts Requirement R4.1.

Initial RAM File System — The initramfs contains a minimal system and an init script that mounts the root file system. Typically, this is not a monolithic script, but a controlling script that delegates control to different scripts for specific use cases, e.g., local and network root file system. The Image Compositor can be placed in this phase by adding such a script to the initramfs that initiates the image composition, and requesting the control script to invoke it. When the Image Compositor is integrated in this way, the resulting initramfs remains generic, i.e., it can boot any virtual machine, whether it uses a composite disk image or not. Alternatively, an initramfs dedicated to image composition can be created that only contains the Image Compositor, at the cost of loosing the flexibility the former approach provides.

Since it is supposed to be flexible, all required kernel modules are very likely already part of the initramfs. Additionally, the minimal system obviously contains a shell for executing the included scripts and a set of useful tools that can be leveraged by the Image Compositor. Additional kernel modules and tools can be added to the initramfs, because the tools that create it are typically extensible.

The options for implementing the Image Compositor are restricted to a native binary or a shell script, unless an execution environment for another language like python is added to the initramfs. But since the initramfs should be as small as possible to prevent a negative impact on boot time, using such a language should be carefully considered.

Init System — As stated in the description of the Linux boot process in Section 4.3.3.1, after the root file system is mounted, the system's root directory is switched to the root file system[2] and the init system takes over control. The image composition must obviously be finished at this point of time, so the init system can begin to start all the configured services that make up the system.

[2] Unless the kernel mounted the root file system itself, without relying on an initramfs. In this case, the system's root directory is already set correctly.

One way to place the Image Compositor in the last phase of the boot process is to schedule it for early execution by the init system, before any other service is started, and to switch the system's root directory again. Unfortunately, it is non-trivial to switch the system's root directory as soon as a running process exists that depends on that root directory. The init system, or more specifically its PID 1 process, the first process started by the Linux kernel[3] and the last to exit during a shut down, is such a process. Moreover, it is a process that cannot be terminated, since it performs essential function for a Linux system. The procedure to nevertheless switch the system's root directory in this kind of situation is described in Section 4.4.6.2.

Another way to place the Image Compositor in this phase is to move the main binary of the init system and replace it with the Image Compositor itself. After the composition is finished, the Image Compositor must switch the system's root directory and transfer control to the actual init system, like the init script of the initramfs does.

Independent of the way selected to integrate the Image Compositor, any tool or language that is available in the base layer of the composite disk image can be used for implementing the Image Compositor. As already stated in the assessment of the Linux kernel integration, relying on applications available in the base layer imposes some requirements on this layer, which contradicts Requirement R4.1. But even if the Image Compositor is implemented as a native binary without any dependencies, the binary itself needs to be part of the base layer, so Requirement R4.1 is not satisfied anyway.

4.3.3.4 Discussion

The assessment results of the three last phases in the Linux boot process with regard Image Compositor placement are summarized in Table 4.4. Clearly, Requirement R4.1 is a critical requirement for the selection of the right phase to integrate the Image Compositor.

Detailed examination of the assessment results shows that the integration into the initramfs is the most promising approach, because it fully satisfies Requirement R4.1 and imposes almost no restrictions regarding the implementation of the Image Compositor. The integration into one of the other two phases either complicates the implementation excessively or introduces dependencies on the base layer that contradict Requirement R4.1. Furthermore, there is no advantage over the integration into the initramfs in the other two approaches that can compensate their drawbacks.

4.3.4 Proposed Solution

In this section a novel approach to image composition is proposed that satisfies all requirements imposed on an Image Compositor in Section 4.3.1.

[3] If an initramfs is used, it is technically not the first process that is started, but it is still executed by the process with the PID 1.

Table 4.4 Summary: Assessment of the Phases for Integration. Summary of the implementation options, obstacles and unsatisfied requirements depending on the phase in the Linux boot process the Image Compositor is integrated into: the Linux kernel (a), the initramfs (b), or the init system (c).

- Implementation options range from a kernel module (a), over native binaries, to shell scripts and even other scripting languages (b, c).
- Requirement R4.1 is difficult to satisfy depending on external tools (a, c) and language used to implement the Image Compositor (c).
- Requirements R4.5, R4.7 and R4.8 might require to use external tools, which contradicts with Requirement R4.1 depending on the phase chosen for the image composition (a).
- Placement in some of the phases generates additional obstacles for the implementation of the Image Compositor (c).

The goal of virtual machine image composition is to build composite disk images for virtual machines using multiple, independent layers. To provide maximum flexibility, union mounts have been chosen as implementation option for the image composition over a block-level redirect-on-write approach. As a result, the image composition has to be done inside the virtual machine itself during the Linux boot process. Potential options for the integration of the Image Compositor have been evaluated in the last section. The integration into the initial RAM file system has been determined as the most suitable approach.

The evaluation has shown that three of the requirements (Requirements R4.5 to R4.7) are only partially satisfied by this approach. The issues caused by using union mounts for image composition are described below together with suggestions to solve those issues.

Requirements R4.5 and R4.7 both deal with passing information from outside the virtual machine to the Image Compositor. This can either be the configuration for the Image Compositor or additional files for injection into the writable layer. There are multiple ways to pass this information and satisfy the corresponding requirements:

Kernel Command Line — Configuration information (Requirement R4.5) can be passed to the Image Compositor using the kernel command line, as long as it can be expressed using the `key=value` syntax.

Configuration Volume — Both configuration information and preseeding archives can be passed to the Image Compositor using a *configuration volume*, i.e., an additional virtual hard disk containing the information to be passed stored at a specific location. Details are given in Section 4.4.3.

Configuration Server — Both configuration information and preseeding archives can be downloaded from a *configuration server* using the Trivial File Transfer Protocol (TFTP) [149]. Network support is typically available in the initial RAM file system to be able to mount network volumes. Details are given in Section 4.4.3.

Requirement R4.6 demands a transparent composition mechanism, both to ensure software compatibility and to prevent software within the virtual machine from interfering with the image composition. As the composition is done inside the virtual machine, the union mount cannot be hidden from any application, as it is visible via the world-readable file `/proc/mounts`. Still, access to the individual layers can be prevented by not moving the mount points of the layers into the composite disk image. Unfortunately, in some cases the mount point of the writable layer needs to be moved to the composite disk image. In these cases, it is possible to directly access and even modify the writable layer. By choosing a mount point inside the `/root` directory and securing it appropriately, e.g., using modes like `0700` or `0770`, direct access by anyone except the root user can be forbidden. More details about this topic are given in Section 4.4.5.

The following minor issues regarding requirement Requirement R4.6 cannot be solved. They do not prevent the use of union mounts in the Image Compositor, since only the root user can actually interfere with the image composition:

- The root user has full write access to the writable layer using the mount point in his home directory. This is no problem, because the root user has full write access to the whole root file system anyway. Application software should not be executed as root user anyway.
- The root user can mount all of the read-only layers a second time and directly access them. Because they are mounted two times, only read access is possible. Additionally, any hypervisor is able to mount disk images read-only, so the root even cannot modify if he could access the original mount point.
- The union mount and all layers are visible at least in `/proc/mounts`. Most application software should not even notice the layers, so this is very likely a minor issue.

An overview of the image composition process is given in Figure 4.13. The process is divided into three distinct stages. The first stage is the *Configuration Stage* that is responsible for evaluating the configuration of the Image Compositor. This optionally includes obtaining an external configuration from a configuration server. In the *Composition Stage* all layers are mounted and a RAM disk is created as writable layer if required. Then, the writable layer is preseeded using files obtained from either a configuration volume or configuration server if needed and the union mount is created. Finally, any composition script found in a specific directory is executed. The *Boot Stage* finishes the image composition by switching the root directory to the composite disk image and transferring control to the init system.

4.4 Implementation

In this section, the implementation of the Image Compositor will be presented. It is structured as follows: Section 4.4.1 gives an overview of the functioning of the initramfs of Debian GNU/Linux that is used as the basis of the implementation. Section 4.4.2 contains an overview of the implementation of the Image Compositor. A detailed

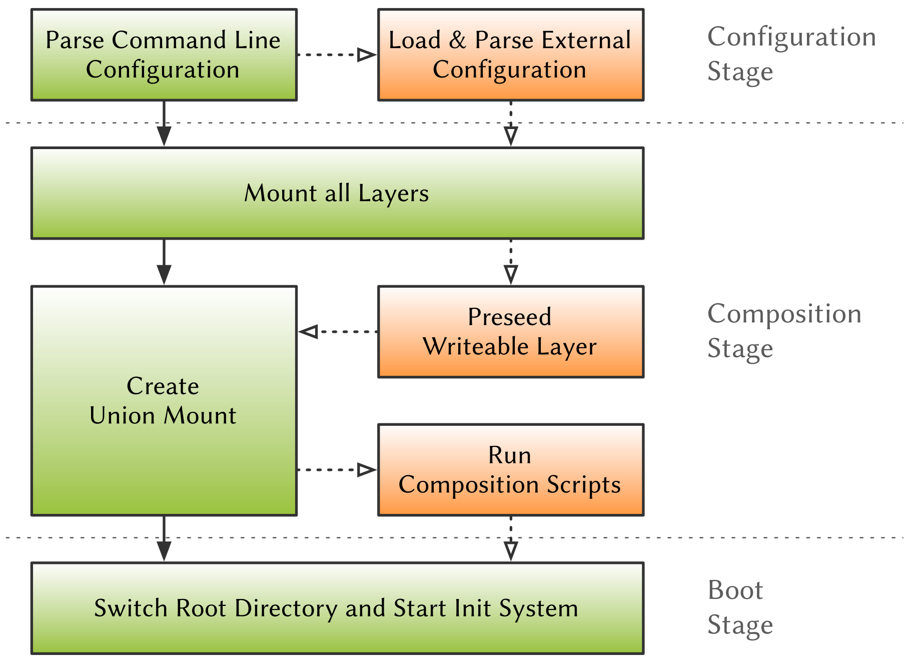


Figure 4.13 Design of the Image Composition Process. An overview of the image composition process consisting of the Configuration Stage, the Composition Stage and the Boot Stage. Orange color denotes optional steps like loading an external configuration file, preseeding the writable layer or running composition scripts.

description of the implementation is given in Sections 4.4.3 to 4.4.5 based on the three phases of the image composition process. Finally, necessary modifications to the system required to solve a few issues caused by the composite disk image are presented in Section 4.4.6.

4.4.1 The Debian-style Initial RAM File System

The implementation of the Image Compositor is based on Debian GNU/Linux 6 (Squeeze), which is not the latest release, but still supported with security updates at the time of writing. As the list of dependencies of the Image Compositor is rather small, it is very likely compatible with newer versions of Debian GNU/Linux. A quick examination of the initramfs of both Debian GNU/Linux 7 (Wheezy) and 8 (Jessie) revealed no fundamental changes. Further testing showed that the Image Compositor works out of the box with both releases[4].

To be able to use the Image Compositor, the system requirements listed in Table 4.5 must be satisfied. Even if a Linux distribution does not satisfy all of the requirements, it might be usable with image composition. The way the Image Compositor is implemented is very modular, so it should be possible to adopt it to other initramfs systems like *dracut* [68] that is used by Fedora. The use of alternative union mount implementations, e.g., its predecessor UnionFS or *OverlayFS* that has been integrated into the

[4] As long as the System V style init system is used in Jessie instead of systemd that is the new default.

Linux kernel in December 2014, is possible but might require minor modifications to the Image Compositor.

Table 4.5 Image Compositor - System Requirements. A Linux system must satisfy the following requirements to be compatible with the Image Compositor.

- The Linux distribution must use a Debian-style initramfs that is described below. This includes the tools that build the initramfs: initramfs-tools.
- The Linux distribution must support the aufs file system.

This section first describes the functioning of the Debian-style initramfs (Section 4.4.1.1) and then explores the options to integrate the Image Compositor into the initramfs (Section 4.4.1.2).

4.4.1.1 Functioning of the Initial RAM File System

In the following, the inner workings of the initramfs used by Debian GNU/Linux and its derivatives like Ubuntu are described.

As outlined in the description of Phase 4 of the Linux boot process (Section 4.3.3.1), after the kernel finished booting, it executes the *init script* in the initramfs that is in charge of mounting the root file system, switches to the root file system and starts the real init system. An overview of the mechanism used to mount to root file system is given in Figure 4.14. The init script starts with mounting the special system directories `/proc` and `/sys` that are essential for almost all of the tools involved in the mounting process. Afterwards, it reads its default configuration that is hard coded into the initramfs and parses the kernel command line that contains import options, e.g., the root device, mount options, or breakpoints.

Before continuing, the first set of hook scripts (*top*) is executed. Among others, one of these hook scripts is starting *udev*, the device manager of the Linux kernel that manages the *device nodes* in the `/dev` directory. These devise nodes are required for mounting the root file system later on. Then, it loads some basic kernel modules, and executes the second set of hook scripts (*premount*).

Now the *boot script* that is responsible for actually mounting the root file system takes over. After the boot script has finished, the last set of hook scripts (*bottom*) is executed. These include scripts that move the `/dev` directory to the root file system and terminate udev that is restarted later on by the init system. The penultimate step is to move the remaining system directories `/proc` and `/sys` to the root file system and to switch the system's root directory to it, as outlined in the description of Phase 5 of the Linux boot process (Section 4.3.3.1). Finally, the real init system is started.

The boot script, or more precisely the `mountroot()` function defined in the boot script, is called from within the init script to do the actual mounting. A default Debian-style initramfs contains two different boot scripts: `local` and `nfs`. The former is able to mount local root file systems, while the latter is able to mount root file systems via NFS. Their structure is similar to that of the init script, containing the same three sets of hook scripts. The following description focuses on the `local` boot script.

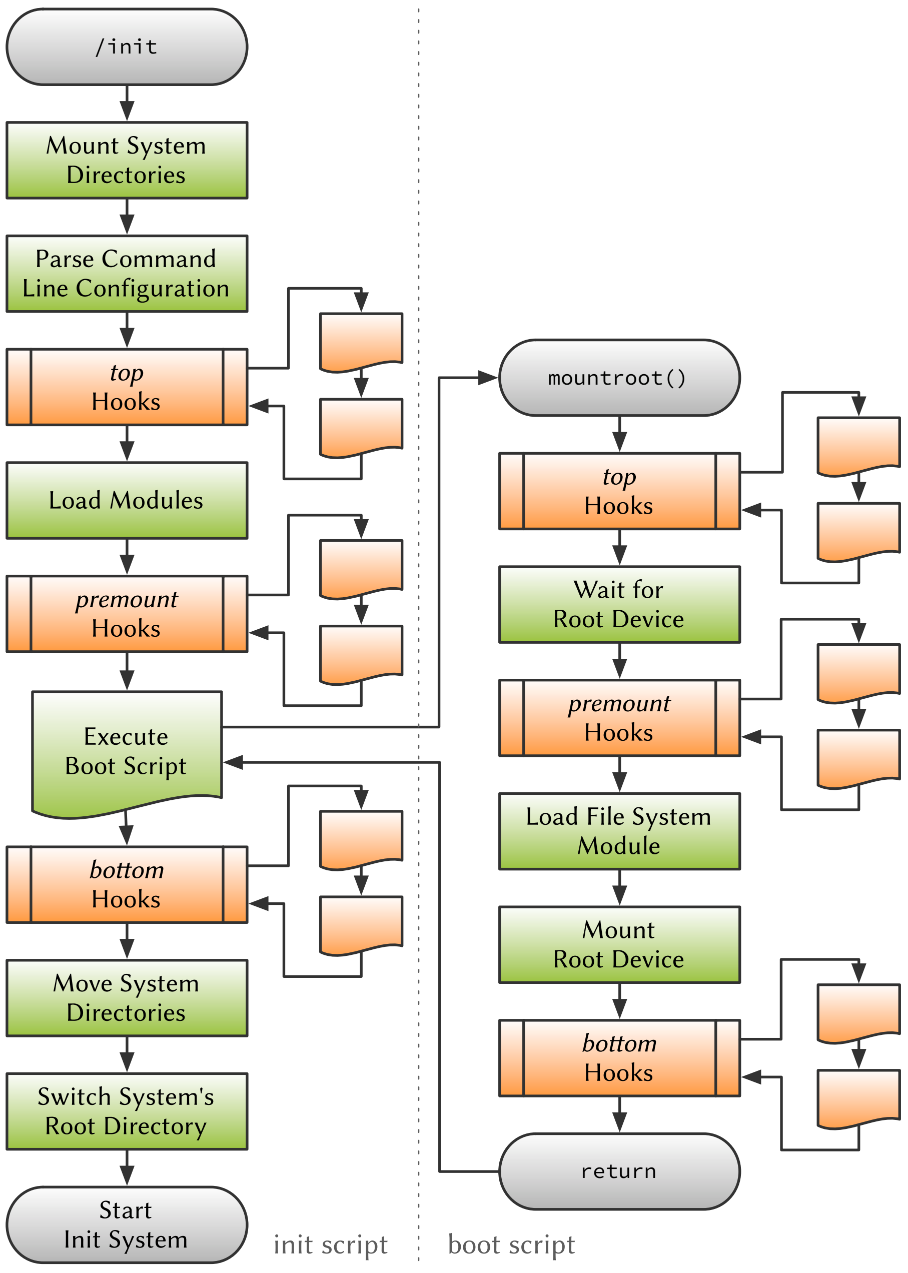


Figure 4.14 Inner Working of the Initial RAM File System. This flow chart describes the process of mounting the root file system in a Debian-style initramfs. It shows both the init script and the selected boot script as well as the hook mechanisms used in both scripts.

After executing the first set of hook scripts, the boot script waits for the device node of the device containing the root file system to show up in /dev, unless it has already appeared. Waiting is necessary, because device nodes are generated dynamically by udev and /dev is empty at the time the init script is started. As soon as the device is available, the seconds set of hook scripts are executed. Then the boot script loads the file system driver kernel module for the root file system, mounts the root device at a predefined mount point, executes the last set of hook scripts, and returns to the caller.

The following snippet shows the parts of a Debian-style initramfs that have been described in the section. Kernel modules, binaries and configuration files have been omitted. The init script is in the root directory of the initramfs, whereas the boot scripts are in the `scripts` directory. For each of the sets of hook scripts, a separate directory exists under `scripts`, except if there are no scripts for a given hook. The directory names are build by combining the name of the script and the name of the set, e.g., `init-top` for the *top* set of hook scripts used by the init script.

```
/
├── init
└── scripts
    ├── functions
    ├── init-bottom
    │   ├── order
    │   └── udev
    ├── init-top
    │   ├── all_generic_ide
    │   ├── blacklist
    │   ├── keymap
    │   ├── order
    │   └── udev
    ├── local
    ├── local-premount
    │   ├── order
    │   └── resume
    ├── local-top
    │   ├── lvm2
    │   └── order
    └── nfs
```

Relevant parts of a Debian-style initramfs.

4.4.1.2 Integration Options

Based on the description in the last section, there are three different ways to integrate the Image Compositor into a Debian-style initramfs:

1. The Image Compositor can be integrated as one of the boot script's bottom hook scripts. In that case, the boot script mounts the root file system as usual, but before it returns to the init script, the Image Compositor takes action. It treats the mounted root file system as lowest layer, mounts all other layers and does the image composition. By relying on the boot script for mounting the first layer, scenarios that combine an NFS mounted root file system with a temporary layer can be realized easily without explicitly supporting NFS. It is important to ensure that the Image Compositor is the first bottom hook script executed, as the other bottom hook scripts might rely on a fully mounted root file system.

2. The Image Compositor can be integrated as a separate boot script. It takes full control of the mounting process, i.e., it mounts all layers and does the image

composition. Features like NFS layers must be explicitly implemented when this option is used, but the resulting Image Compositor is more flexible, e.g., by supporting multiple NFS layers instead of only the base layer.

3. The Image Compositor can replace the init script altogether. In that case, it has to take care of the complete process described in the last section, including the functionality provided by the hook scripts.

Of the three ways to integrate the Image Compositor into the Debian-style initramfs, option 3 is the least suited one, because large parts of the init script need to be reimplemented and the resulting initramfs would be less flexible, i.e., it could not be used anymore to boot a virtual machine without a composite disk image. Option 1 perfectly integrates into the initramfs because it uses the existing hook mechanism, but suffers from the problem of execution order, because the Image Compositor is not a regular bottom hook script, but provides functionality that is actually better placed into the boot script itself. Thus, option 2 is the best choice.

4.4.2 The Composition Process

In this section, the composition process implemented by the Image Compositor as boot script for a Debian-style initramfs is described in detail (Figure 4.15). This description is based both on the design of the Image Compositor (Section 4.3.4) and the functioning of the initramfs in Debian GNU/Linux (Section 4.4.1). For reference, the requirements satisfied by each of the individual steps are noted in parentheses. Requirements R4.1 and R4.6 are satisfied implicitly by the technologies chosen in the design (Sections 4.3.2 and 4.3.3).

The Image Compositor is invoked by the init script through a call to the `mountroot()` function. The first step in the image composition process is the evaluation of the kernel command line, to extract the relevant parameters for the Image Compositor. Unfortunately, the kernel command line can only for be used for controlling the Image Compositor in a few cases: either with Xen PV virtual machines (except in combination with PV-GRUB) or with KVM or Xen HVM virtual machines in direct kernel boot mode. In all other cases, the kernel command line cannot be easily modified when a virtual machine is started, because it is stored in the configuration of the boot loader on the virtual hard disk. By using an external configuration for the Image Compositor (see Section 4.4.3), the parameters can be controlled from the outside with any hypervisor. If an external configuration exists, the Image Compositor loads it and extracts the parameters it contains. Thereby, the parameters from an external configuration overwrite the current parameters (Requirement R4.5).

Afterwards, all read-only layers are mounted one by one, at dynamically created mount points inside the initramfs (Requirements R4.2 and R4.3). If additional kernel modules are required for mounting, either drivers or file system support, they are loaded automatically. The writable layer on top of the composite disk image is either a hard disk based layer or a RAM disk, which is mounted or created, respectively, depending on the configuration (Requirement R4.4). If required, the writable layer can then be preseeded, which is especially useful in combination with a RAM disk, but

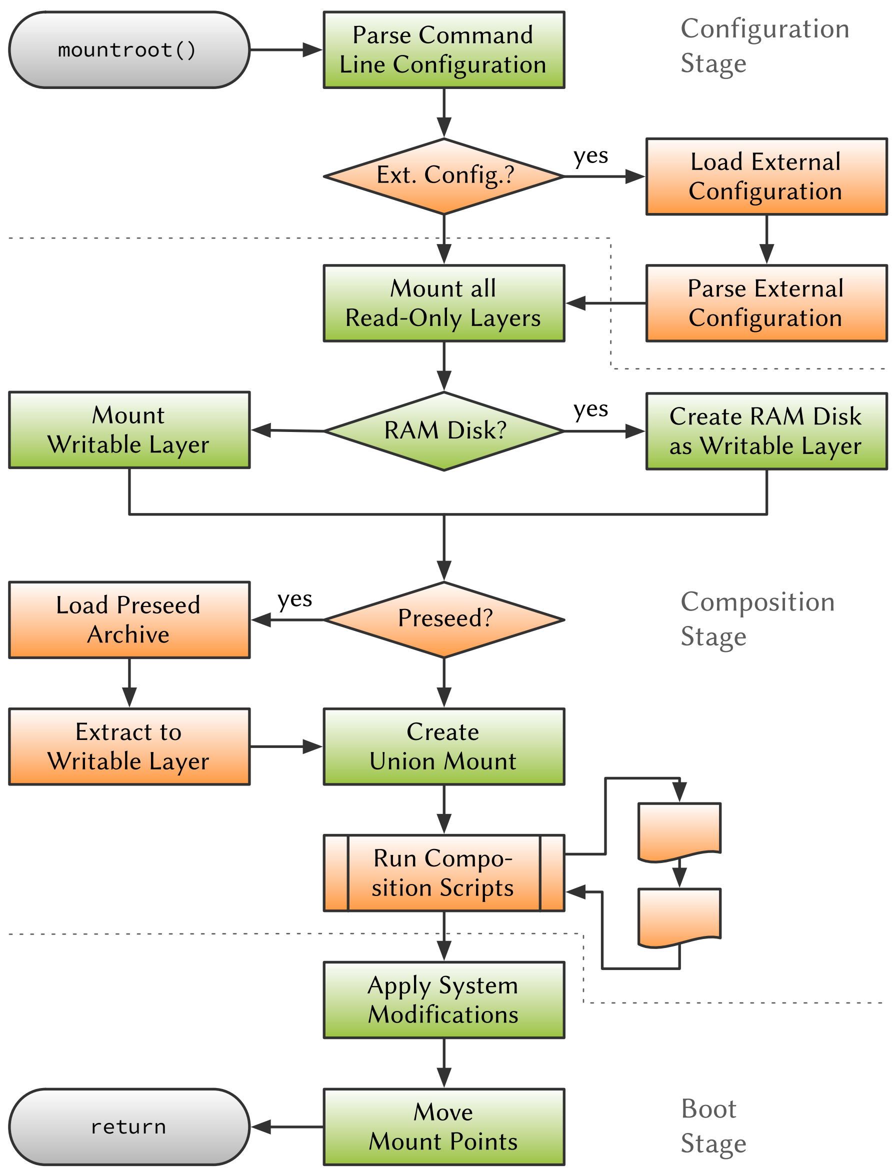


Figure 4.15 The Composition Process in Detail. This flow chart describes the image composition process. It is a more detailed version of Figure 4.13 that depicts the integration as boot script. The green parts of the chart are the essential steps required for image composition: parsing the configuration, mounting the layers, and creating the union mount. The orange parts are optional steps: loading an external configuration, preseeding the writable layer, and running composition scripts.

also possible for hard disk based layers. Preseeding happens by loading the preseed archive an extracting it to the writable layer (Requirement R4.7).

In the next and most important step of the image composition, the union mount is created from the layers mounted in the last steps. The resulting root file system is mounted at a predefined mount point, so the init script can properly switch the system's root directory and transfer control to the init system later on. After the root file system is mounted, the Image Compositor optionally executes composition scripts that are either contained in one of the layers or were added to the root file system using the preseeding mechanism. Composition scripts enable the user to make arbitrary changes to the root file system before the actual init system is started (Requirement R4.8).

If an unmodified Debian GNU/Linux 6 system is booted in this state, some error messages would likely be generated during the boot process. These errors are caused by the fact that some of the scripts the init system uses to start the system are not compatible to composite disk images. Another issue occurs during shutdown, if the writable layer needs to be unmounted correctly. This is of special importance during the creation or modification of a layer, when it is intentionally modified (see Figure 4.5). These issues and their corresponding solutions that are based on modifying configuration files and parts of the init system are described in more detail in Section 4.4.6. The penultimate step of the image composition process is to apply these solutions. Technically, these modifications can be done by a special set of composition scripts, if these scripts are guaranteed to be executed at the end of the composition process.

Finally, the mount points of the individual layers can be moved into the root file system, into a hidden folder below `/root`, where users except the *root* user cannot access them. This step is only required for the writable layer, in case it needs to be unmounted correctly (see Section 4.4.6.2).

4.4.3 Configuration Stage

In the configuration stage, the Image Compositor reads its configuration that can be stored in the initial RAM file system (hard-coded) or provided using either the kernel command line or an external configuration source. The Image Compositor recognizes eleven parameters: six parameters that influence the composite disk image and its composition process (see Table 4.6) and five parameters that are useful for troubleshooting the composition process (see Table 4.7).

The kernel command line is not only used to control the boot process, but to configure many different subsystems of the Linux kernel. This widespread usage of the kernel command line calls for the use of name spaces to prevent clashes between the parameters for different subsystems. All Image Compositor parameters are prefixed with `crfs`. An exemplary kernel command line is given below.

```
boot=crfs crfs.rlayers=/dev/vda1,/dev/vdb1 crfs.wlayer=tmpfs          ⤸
··· crfs.tmpfssize=1024M crfs.verbose
```

An exemplary kernel command line for using the Image Compositor.

The first parameter `boot=crfs` is a parameter evaluated by the init script. It selects the boot script responsible for mounting the root file system. The remaining parameters are parameters to the Image Compositor. In this specific case, a composite disk image is built from three layers, two read-only layers based on the first two virtual hard disks

Table 4.6 Image Compositor Parameters – Composite Disk Image. The following set of parameters influences both the composition process and the resulting composite disk image.

Parameter	Description
`config=<arg>`	The source of external configuration: a device node, the keyword *auto*, or a TFTP URL. Optional parameter. Examples: `config=tftp://tftp.local/crfs/config` or `config=auto`
`rlayers=<arg>`	The read-only layers that are part of the composite disk image: a comma-separated list of one or more device nodes, starting with the lowest layer in increasing order. Example: `rlayers=/dev/vda1,/dev/vdb1`
`wlayer=<arg>`	The writable layer of the composite disk image: a device node or the keyword *tmpfs*. Optional parameter, if not set, *tmpfs* is used. Examples: `wlayer=/dev/vdc1` or `wlayer=tmpfs`
`tmpfssize=<arg>`	The size of the RAM disk used as writable layer: an integer value including unit (*K*, *M*, or *G*). Optional parameter. Example: `tmpfssize=1024M`
`preseed=<arg>`	The source of a preseed archive: a device node, the keyword *auto*, or a TFTP URL. Optional parameter. Examples: `config=tftp://tftp.local/crfs/preseed` or `config=/dev/vdc1`
`scripts=<arg>`	The layer that contains the composition scripts: a device node or the keyword *tmpfs*. Optional parameter. Examples: `scripts=/dev/vdc1` or `scripts=tmpfs`

– more specifically the first partition of each of the first two virtual hard disks – and a writable layer based on a RAM disk of 1024 MiB. Neither an external configuration is used, nor preseeding or composition scripts. The Image Compositor is put into verbose mode to output more information about the composition process.

The part of the `mountroot()` function that is shown in Listing 4.1 corresponds to the configuration stage as depicted in Figure 4.15. The functions `crfs_trace` and `crfs_maybe_break` are used for logging and breakpoints, respectively. The call to `wait_for_udev` in Line 4 ensures that udev has finished preparing the device nodes for the devices in the system. The actual composition starts in Line 7 with loading hard coded default values from the initramfs (not shown in Figure 4.15), if the corresponding file `/etc/crfs` exists. This facilitates the creation of an initramfs with specific settings for an environment, without relying on the kernel command line. The next step is to parse the kernel command line using the `crfs_parse_cmdline` function that reads the file pointed to by the first argument and parses the arguments it contains. The Linux kernel command line is always available as `/proc/cmdline`, so it is passed as argument to the function.

After the kernel command line has been parsed, the Image Compositor checks whether

Table 4.7 Image Compositor Parameters – Troubleshooting. The following set of parameters can be used for finding and fixing problems in the image composition process.

Parameter	Description
`verbose`	Log detailed progress information to the console.
`no_break_on_error`	Halt the system after errors instead of opening a shell for debugging. Useful in production.
`breakpoint=<`*`arg`*`>`	Break the composition process at the given breakpoint and open a shell for debugging. Possible breakpoint values: 1 – 10. Example: `breakpoint=4`
`stepping`	Break the composition process before each step and open a shell for debugging.
`trace`	Log debugging information to the console.

```
1  mountroot () {
2    crfs_trace "mountroot"
3
4    wait_for_udev 10
5
6    # load hard-coded settings
7    [ -f /etc/crfs ] && source /etc/crfs
8
9    crfs_parse_cmdline /proc/cmdline
10
11   crfs_maybe_break 1 "loading external config"
12
13   if [ -n "$crfs_config" ]
14   then
15     local old_config=$crfs_config
16     crfs_load_external_config
17     if [ "$old_config" != "$crfs_config" ]
18     then
19       # the external config redirected to another source
20       crfs_load_external_config
21     fi
22     unset old_config
23   fi
24
25   crfs_maybe_break 2 "validating config"
26
27   crfs_check_config
```

Listing 4.1 Configuration Stage. The first part of the `mountroot()` function that corresponds to the configuration stage as depicted in Figure 4.15. It is responsible for loading and evaluating the configuration of the Image Compositor.

an external configuration should be loaded (Line 13). The variable `crfs_config` contains the value of the `config` parameter, i.e., the source that should be used to find the external configuration: either a device node, the keyword *auto*, or a TFTP Uniform Resource Locator (URL). In the first two cases, a *configuration volume* is mounted that must contain a file name `cmdline` in the root directory. The format of this file has to be the same as the file `/proc/cmdline`, because the same function is used to parse the parameters. In the case of the *auto* keyword, the device node of the configuration volume is automatically determined by searching for a partition with a specific Universaly Unique Identifier (UUID).

```
43524653-0000-0000-0000-434f4e464947
```

The UUID of Image Compositor configuration volumes.

If the variable contains a TFTP URL, the file pointed to by the URL is downloaded and parsed exactly like a `cmdline` file in a configuration volume.

The old value of the variable is saved before an external configuration is loaded via `crfs_load_external_config` (Line 15). This facilitates chain-loading a second external configuration, if the `config` parameter is changed by the external configuration (Lines 17 to 21). A configuration process like the following is enabled by chain-loading of external configuration:

1. The hard coded configuration in the initramfs contains the `config=auto` parameter. This instructs the Image Compositor to search for a configuration volume on all virtual hard disks.
2. The `cmdline` file on the configuration volume contains a TFTP URL as `config` parameter. This instructs the Image Compositor to fetch the actual configuration from the specified server.
3. The downloaded file contains the actual configuration of the Image Compositor.

Using this approach, a standard initramfs can be used to load the configuration from any server, by attaching a virtual hard disk with an appropriate configuration volume. This facilitates very easy adaption of the configuration by changing the file on the TFTP server. Even if the TFTP server changes, only the configuration volume needs to be replaced, but not the initramfs of every virtual machine[5].

The final step of the configuration process is to call the `crfs_check_config` function. It verifies the existence of all layers referenced in the configuration and sets default values if required, e.g., a RAM disk as writable layer if none is defined or a default size for RAM disks.

4.4.4 Composition Stage

The composition stage is the primary part of the image composition process. It consists of mounting all layers, optionally preseeding the writable layer, creating the union mount, and optionally running composition scripts (see Figure 4.15).

The part of the `mountroot()` function shown in Listing 4.2 is responsible for mounting the read-only layers. It consists of a for loop that iterates over all of the read-only

[5] Because the initramfs contains the kernel modules for devices and file systems not compiled directly into the kernel, each kernel version requires a distinct initramfs.

layers set using the `rlayers` parameter. The `crfs_check_config` function (called in Line 27 of Listing 4.1) took care of converting the comma-separated list into a whitespace-separated list that can be used with the for loop.

```
1  crfs_maybe_break 3 "mounting readonly layers"
2
3  # in this variable the aufs dirs argument (the layer hierarchie) is built
4  local crfs_unionconfig=
5  # temporary variables for iterating through all layers
6  local layer mountpoint
7  for layer in $crfs_rolayers
8  do
9    crfs_next_mountpoint
10   mountpoint=$(crfs_mountpoint)
11   # mount writable first, to see if the hypervisor blocks write requests
12   if crfs_mount_fs "$layer" "$mountpoint" "rw"
13   then
14     if crfs_is_writable "$mountpoint"
15     then
16       crfs_warning "write access to read-only layer $layer not prevented"
17       # this can be ignored: once a device is mounted ro, it can not be
18       # mounted a second time with rw (only again with ro)
19       mount -o remount,ro "$mountpoint" || crfs_fatal "remount ro failed"
20     fi
21     if [ -z "$crfs_unionconfig" ]
22     then
23       crfs_unionconfig="$mountpoint=ro"
24     else
25       crfs_unionconfig="$mountpoint=ro:$crfs_unionconfig"
26     fi
27   else
28     crfs_error "failed to mount $layer"
29   fi
30 done
```

Listing 4.2 Composition Stage - Mount Read-only Layers. The second part of the `mountroot()` function that starts the composition stage as depicted in Figure 4.15. This part mounts all read-only layers that are part of the composite disk image.

The function `crfs_next_mountpoint` dynamically crates a new mount point for each layer (Line 36). These mount points are named `/.crfslayerN`, where `N` is the number of the layer. Because shell functions cannot return values other than integers, the name of the newly generated mount point is printed to the standard output by the `crfs_mountpoint` function. Using command substitution, the mount point is saved in the corresponding variable (Line 37).

The layer is then mounted at that mount point using the `cref_mount_fs` function (Line 37). The Image Compositor first tries to mount each read-only layer in writable mode, to test if the hypervisor does prohibit write access. If the layer can be mounted writable, a warning is issued and the layer is remounted read-only (Lines 43 and 46). Although the configuration of a layered virtual machine should not allow write access to read-only layers, the composition process does not need to be aborted at this point. Since the layer is part of the union mount, there is no way to mount it writable a second time, so the layer cannot be modified during runtime of the virtual machine. Nevertheless, it is technically possible to modify the layer after unmounting the union mount during the shutdown procedure (see Section 4.4.6.2), so the configuration of the virtual machine should be changed accordingly.

The variable `crfs_unionconfig` is used to build the configuration for aufs (Lines 48 to 53). The string is used as `dir` option when then union mount is created. It contains a list of mount points and access modes, separated by colons, with the topmost layer first and the bottom layer last (the inverse order compared to the `rlayers` parameter of the Image Compositor).

The part of the `mountroot()` function shown in Listing 4.3 is responsible for mounting the writable layer. The writable layer is stored in the `crfs_rwlayer` variable that corresponds to the `wlayer` parameter. Possible values for the writable layer are a device node or the keyword *tmpfs* for a virtual hard disk or a RAM disk based writable layer, respectively. In both cases, the `crfs_mount_fs` function is used to mount the layer (Line 69). In case of a RAM disk, its size is stored in the `options` variable beforehand (Lines 64 to 67). The value of the `tmpfssize` parameter has been stored in the required format in the `crfs_tmpfs_options` variable by the `crfs_check_config` function (called in Line 27 of Listing 4.1).

```
31    crfs_maybe_break 4 "mounting writeable layer"
32
33    crfs_next_mountpoint
34    mountpoint=$(crfs_mountpoint)
35    layer=$crfs_rwlayer
36    local options=rw
37    if [ "$layer" = "tmpfs" ]
38    then
39      options=$crfs_tmpfs_options
40    fi
41    crfs_log "mounting $layer on $mountpoint ($options)"
42    crfs_mount_fs "$layer" "$mountpoint" "$options"
43    if [ $? -eq 0 ]
44    then
45      if crfs_is_writable "$mountpoint"
46      then
47        crfs_unionconfig="$mountpoint=rw:$crfs_unionconfig"
48      else
49        crfs_fatal "writable layer $layer can not be mounted writable"
50      fi
51    else
52      crfs_fatal "failed to mount writable layer"
53    fi
```

Listing 4.3 Composition Stage - Mount Writable Layer. The third part of the `mountroot()` function that proceeds the composition stage as depicted in Figure 4.15. This part either mounts the writable layer or creates a RAM disk for use as writable layer.

It is important to check if the writable layer is in fact writable (Line 72). While a RAM disk is writable in any case[6], a virtual hard disk based layer might not be writable if the virtual machine configuration is erroneous. The final step is to update the `crfs_unionconfig` variable by inserting the mount point and access mode of the writable layer at the beginning of the string (Line 74).

The part of the `mountroot()` function shown in Listing 4.4 is responsible for both preseeding the writable layer and creating the union mount. The preseeding is done by the `crfs_preseed` function (Line 85), if the variable of the same name that corresponds to the `preseed` parameter contains a valid source for preseeding. Like the external configuration, a valid source can either be a device node, the keyword *auto*, or a TFTP URL. In the first two cases a volume is mounted that must contain a preseed archive in the root directory. The preseed archive has to be a tar archive that can optionally be compressed using either *gzip* or *bzip2*. It has to be named `preseed` and have the correct extension for the archive type (`.tar`, `.tar.gz`, `.tgz`, `.tar.bz`, or `.tbz`). Both external configuration and the preseed archive are supposed to be stored in the same configuration volume, so the same UUID is used to identify the volume containing the preseed archive (see Section 4.4.3).

```
54    crfs_maybe_break 5 "preseeding a tmpfs"
55
56    if [ -n "$crfs_preseed" ]
57    then
58      crfs_preseed "$mountpoint"
59    fi
60
61    crfs_maybe_break 6 "mounting aufs"
62
63    crfs_log "mounting aufs with params '$crfs_unionconfig'"
64    crfs_load_module aufs || crfs_fatal "failed to load aufs module"
65    if ! mount -t aufs -o dirs="$crfs_unionconfig" aufs "$rootmnt"
66    then
67      crfs_fatal "failed to mount aufs"
68    fi
```

Listing 4.4 Composition Stage - Preseed and Create the Union Mount.
The fourth part of the `mountroot()` function that proceeds the composition stage as depicted in Figure 4.15. This part preseeds the writable layer, if requested, and creates the actual union mount.

If the variable contains a TFTP URL, the preseed archive pointed to by the URL is downloaded. It can be arbitrarily named, but has to have the correct extension for the archive type. Otherwise, there is no difference between downloading the preseed archive or loading it from a local volume.

Creating the union mount is straightforward at this point (Lines 91 to 92). First, the aufs kernel module has to be loaded using the `crfs_load_module` function. If this fails, the composition process is aborted. Afterwards, the union mount is created using the mount command. The `dirs` option is used to configure the union mount aufs with the layer hierarchy and access modes.

[6] Except if the RAM disk is explicitly mounted read-only.

The part of the `mountroot()` function shown in Listing 4.5 is responsible for running composition scripts. The variable `crfs_scripts` that corresponds to the `scripts` parameter contains the layer that contains the compositions scripts to execute. For security reasons, only the scripts in a single layer are executed. This ensures that only scripts targeted at the specific environment are executed. Composition scripts in other layers are not executed, because they are not targeted at the environment the virtual machine is executed in and might even be malicious.

```
  crfs_maybe_break 7 "executing composition scripts"

  if [ -n "$crfs_scripts" ]
  then
    case $crfs_scripts in
      /dev/*)
        layer=$crfs_scripts
        [ -b "$layer" ] || crfs_fatal "invalid source of composition scripts"
        mountpoint=$(grep "^$layer" /proc/mounts | cut -d' ' -f2)
        [ $? -eq 0 ] || crfs_fatal "composition script source not mounted"
        ;;

      /tmpfs)
        if [ "$crfs_rwlayer" = "tmpfs" ]
        then
          [ -n "$crfs_preseed" ] || crfs_fatal "no preseeding of composition scripts"
        else
          crfs_fatal "no tmpfs layer used"
        fi
        ;;
    esac

    crfs_log "executing composition scripts from $layer"
    [ -d "$mountpoint/.crfs_scripts" ] || crfs_fatal "no composition scripts found"
    crfs_execute_scripts "$mountpoint/.crfs_scripts"
  fi
```

Listing 4.5 Composition Stage - Run Composition Scripts. The fifth part of the `mountroot()` function that completes the composition stage as depicted in Figure 4.15. This part is responsible for executing the composition scripts, if required.

Valid sources are either virtual hard disk based layers, specified via their device node, or a RAM disk that is used as writable layer, specified via the keyword *tmpfs*. In the former case, the mount point of the layer is determined for valid device nodes (Line 102 to 105). In the latter case, the mount point is still set correctly (Line 61 in Listing 4.3). Obviously, using the keyword *tmpfs* as source only makes sense when both a RAM disk is used as writable layer and the RAM disk was preseeded (Lines 109 to 114).

After the mount point of the source of composition scripts has been selected, the Image Compositor checks whether a directory named `.crfs_scripts` does exist in the root directory of that source (Line 119). This directory contains all composition scripts to be executed. The scripts need to be named according to the scheme `crfs_NN_NAME`, where `NN` is a two-digit number and `NAME` is an arbitrary name. The two-digit number determines the order of execution: scripts with lower numbers are executed first. If multiple scripts with the same number exist, they are executed in the alphabetical order of the `NAME` part. The execution of the composition scripts is controlled by the function `crfs_execute_scripts` (Line 120). Each script is started with a single argument: the path of the union mount.

4.4.5 Boot Stage

In the boot stage, the Image Compositor takes care of the final steps necessary before the system can continue to boot using the composite disk image. These steps consist in modifying parts of the operating system and relevant configuration files to ensure a working system and moving the mount points of the layers from the *rootfs*[7] to the actual root file system. The corresponding code is shown in Listing 4.6.

```
  crfs_maybe_break 8 "applying necessary configuration and system changes"

  crfs_log "applying necessary configuration and system changes"
  crfs_execute_scripts /scripts/crfs.d

  crfs_maybe_break 9 "moving the layer mountpoints"

  # move the mountpoints
  # only the writable layer mountpoint needs to be moved, so it can be
  # cleanly unmounted or cleared before shutdown
  local new_mountpoint="$rootmnt/root/.crfs_wl"
  crfs_log "moving $mountpoint to $new_mountpoint"
  crfs_create_dir "$new_mountpoint"
  if ! mount -o move "$mountpoint" "$new_mountpoint"
  then
    crfs_fatal "moving the writable layer failed"
  fi

  crfs_maybe_break 10 "leaving $NAME"
}
```

Listing 4.6 Boot Stage. The last part of the `mountroot()` function that corresponds to the boot stage as depicted in Figure 4.15. It is responsible for modifying the system to make it compatible with composite disk images and moving the mount point of the writable layer into the composite disk image.

The changes to the configuration and the operating system are dependent on the specific version of the operating system used. An exemplary set of changes is described in Section 4.4.6. A set of scripts stored in the `/scripts/crfs.d` directory of the initramfs is used to implement the required changes. The function `crfs_execute_scripts` that is used to execute composition scripts is also used for these scripts. Accordingly, the same requirements regarding the naming of the individual scripts apply here as well. Like the composition scripts, each of the scripts is invoked with the path of the union mount as its sole argument (see Section 4.4.4).

[7] The RAM file system filled with the contents of the initramfs that is used as temporary root file system during Phase 4 of the Linux boot process (see Section 4.3.3.1).

If the mount points of the layers are not moved from the rootfs to the actual root file system, the layers become inaccessible after the system's root directory has been switched. As a consequence, it is not possible to directly access a layer's contents or to unmount a layer during shutdown. The union mount is fortunately not affected by restriction and continues to work even when the layers are not moved. So the drawback of inaccessible mount points is actually an advantage that prevents tampering with the layers. As long as all virtual hard disk based layers building a composite disk image are read-only, the mount points can be left in the rootfs as read-only mounts do not need to be unmounted.

Unfortunately, not moving the mount points of a writable virtual hard disk based layer prevents a clean unmount, and thus leaves the layer in a broken state. Especially when the writable layer is deliberately modified it (see Figure 4.5) a clean unmount is desirable. Thus, at least the mount point of the writable layer needs to be moved to the actual root file system. This is done in Lines 132 to 138. The old mount point is still stored in the `mountpoint` variable. The writable layer is moved to `/root/.crfs_wl`, where it is protected from accidental modification.

4.4.6 System Modifications

As already stated above, an unmodified operating system might not be able to boot using a composite disk image. The reasons are peculiarities of the union mount used as root file system. In this section, the necessary changes for a Debian GNU/Linux 6 system are presented both for the boot and the shutdown process.

4.4.6.1 Boot Process

There is a single init script in the boot process of Debian GNU/Linux 6 that is not compatible with a composite disk image: the `checkroot.sh` script. Its task is to check the root file system for file system errors that might occur if the virtual machine is not shut down cleanly. While this a very reasonable thing to do for virtual machines that use a standard root file system, it is not required if a composite disk image is used. Most layers are mounted read-only anyway, so corruption is very rare and can only occur if a layer is modified (see also Section 4.4.6.2). Moreover, if `checkroot.sh` actually tries to check a composite disk image it will fail with a non-recoverable error and restart the system.

The root file system is checked if its *pass* option in the file system table `fstab` that controls the order of file system checks during boot is set to a value other than zero. By default, the value is set to 1, so the root file system is the first to be checked. Another problem can occur if the file system table contains mount *options* that are not compatible with aufs. One of these problematic options is `errors=remount-ro` that is set by default for the root file system in Debian GNU/Linux 6. While it does not prevent the system from booting successfully, it should be removed from the *options*. The following snippet shows the structure of the file system table and a few exemplary entries.

```
# <file system>           <mount point>  <type>  <options>  <dump> <pass>
UUID=00b62ed8-fc(...)df   /              ext3    errors=remount-ro  0  1
UUID=90487b4e-25(...)9a   none           swap    sw                 0  0
proc                      /proc          proc    defaults           0  0
```

Exemplary entries from a file system table: a root file system, a *swap* partition and the *proc* file system. (Indentation has been modified for easier reading.)

The script `crfs_10_patch_fstab` (Listing 4.7) creates a backup of the file system table and modifies the entry of the root file system appropriately. It does so using the *awk* utility that is part of the initramfs as part of *busybox*. The corresponding *AWK* program is stored in an external file named `patch_fstab.awk` (Listing 4.8). It is invoked in Line 25, reading the backup file and outputting the patched file system table.

```
15 FSTAB=$ROOT/etc/fstab
16 BACKUP=$FSTAB.crfs
17
18 # make a backup of the old fstab, unless it already exists
19 # don't exit if it exists, instead patch it again from the original fstab
20 if [ ! -f $BACKUP ]
21 then
22   cp $FSTAB $BACKUP
23 fi
24
25 awk -f /scripts/crfs.d/patch_fstab.awk $BACKUP >$FSTAB
```

Listing 4.7 Script that Patches the File System Table. This script named `crfs_10_patch_fstab` sets the *pass* option of the root file system in the file system table to zero and removes `errors=remount-ro` from the *options* using the AWK program shown in Listing 4.8.

The AWK program consists of two action blocks. The first action block (Lines 1 to 4) is preceded by a pattern that matches both comments starting with the # character and all entries except the root file system. The latter are identifiable by a mount point unequal to `/`. This action block outputs the corresponding lines unchanged and uses the `next` statement to finish the processing of those lines.

The second block (Lines 6 to 12) is not preceded by any pattern and is thus executed for all remaining lines. By design, the only remaining line is the entry of the root file system. In Line 7 the *pass* option, which is the 6th field, is set to zero. The call to the `sub` function in Line 8 removes `errors=remount-ro` and optionally a subsequent comma from the *options* (4th field). If *options* is empty after removal the file system table cannot be parsed anymore, because it is not a fixed-column-aligned but a whitespace-delimited file. Typical parsing code would therefore interpret the *dump* option as *options* and the *pass* option as *dump*. Consequently, *options* must not be empty. In Line 9, `rw` is stored in *options* in this case. As the composite disk image is already mounted writable this option cannot cause any problems. Finally, the tab character is set as field separator and the updated line is outputted.

```
($1 ~ /^#/) || ($2 != "/") {
    print $0;
    next
}

{
    $6=0;
    sub(/errors=remount-ro,?/, "", $4);
    sub(/^$/, "rw", $4);
    OFS="\t";
    print $0;
}
```

Listing 4.8 AWK Program to Patch the File System Table. This is the AWK program named `patch_fstab.awk` used by the `crfs_10_patch_fstab` script to modify the file system table of a Debian GNU/Linux 6 in order to make it boot from a composite disk image.

4.4.6.2 Shutdown Process

There are two issues that need to be taken care of during the shutdown procedure. The first issue appears when a RAM disk is used as temporary layer. During shutdown, the swap is disabled as one of the last steps before the root file system is unmounted. All RAM disks are unmounted before this step, because RAM disks might use more memory than physically available. Both steps are part of the `umountfs` script in Debian GNU/Linux 6. If a RAM disk is part of the composite disk image, it cannot be unmounted. Therefore it has to be cleaned before the swap is disabled.

The second issue appears when the composite disk image is configured without a temporary layer. In this case, write operations are conducted on the writable layer (the highest one) of the composite disk image (see Figure 4.5). Since these modifications are intentional, the integrity of the writable layer has to be preserved during the shutdown procedure. Thus, the composite disk image needs to be cleanly unmounted. With temporary layers, this is not required, since all persistent layers are read-only anyway. Unfortunately, the default `umountroot` script in Debian GNU/Linux 6 is not able to cleanly unmount a composite disk image.

Figure 4.16 shows the last steps of the standard shutdown procedure in Debian GNU/Linux 6. When a shutdown command is issued, the runlevel – a system state associated with a specific combination of running services – is changed to 0 (halt) and the `/etc/init.d/rc` script is invoked. Its duty is to start and stop services using init scripts whenever the runlevel is changed. When switching to 0, it stops all services because in runlevel 0 no services are active. Finally, it invokes a number of init scripts in `/etc/rc0.d` that take care of shutting down the system correctly. The last three of them are shown in the figure. The script `umountfs` takes care of unmounting all RAM disks, swap devices and local file systems. Afterwards, the `umountroot` script remounts the root file system read-only. Finally, the `halt` script shuts down the machine[8].

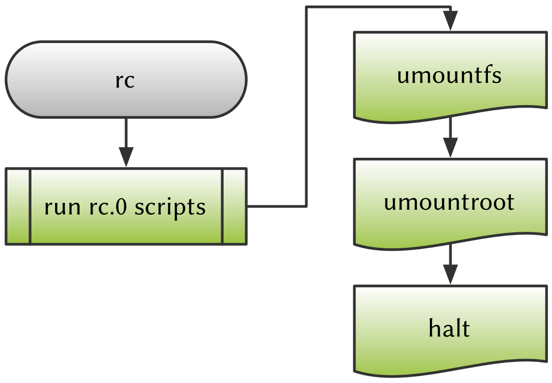


Figure 4.16 The Shutdown Procedure of a Linux System. This flow chart shows the last steps in the shutdown procedure of an unmodified Debian Squeeze: the `rc` script executes the last three scripts in `/etc/rc0.d/` that unmount all file systems, mount the root file system read-only and halt the system, respectively.

Contrary to its name, the `umountroot` script does not unmount the root file system. The reason is that a file system can only be unmounted when there is no process referencing it left running. As each process in Linux stores a reference to the root file system it cannot be easily unmounted. During this part of the shutdown process, at least the init process and two shells executing the `rc` and `umountroot` scripts are running and thus referencing the root file system, preventing it from being unmounted. Fortunately, remounting a file system read-only serves the same purpose: writing all remaining caches to disk, closing the journal and putting the file system in an unmodifiable state.

The shutdown procedure described in the last paragraph fails for a composite disk image without a temporary layer. While the `umountroot` script does remount the root file system read-only, the writable layer itself is still mounted in writable mode. So when the machine is shut down afterwards by the `halt` script[9], some caches might not have been flushed and the journal is not closed. Thus, a file system recovery is required on the writable layer before it can be used again.

The first issue can be solved in two steps. The first step is to ensure that the `umountfs` script does unmount a RAM disk used as writable layer[10]. This can be done by adding the path of the writable layer to a list of mount points that are ignored by the `umountfs` script. The script `crfs_20_patch_umountfs` shown in Listing 4.9 takes care of this.

The `sed` command in Line 11 first searches the line with the list of mount points that starts with `/|/proc`. In this line, it appends the mount point `/root/.crfs_wl` to

[8] For a reboot, the respective runlevel would be 6 and the last script to be executed would be the `reboot` script.

[9] Or rebooted by the `reboot` script.

[10] Trying to unmount a RAM disk that is part of a composite disk image would fail anyway for the reasons stated above.

```
UMOUNTFS=$ROOT/etc/init.d/umountfs

# if already patched - do nothing
grep -F -qs '|/root/.crfs_wl)' $UMOUNTFS && exit 0

# patch /etc/init.d/umountfs to ignore the writable layer
sed -i -e '/\/|\/proc|/ s/)$/|\/root\/.crfs_wl)/' $UMOUNTFS
```

Listing 4.9 Script Patching the umountfs Init Script. This script named `crfs_20_patch_umountfs` adds the mount point of the writable layer inside the composite disk image (`/root/.crfs_wl`) to a list of mount points that are ignored by the `umountfs` init script, i.e., not unmounted during shutdown.

the list of conditions of the case branch that is used to skip certain special mount points.

The next step is to install a script that takes care of cleaning the RAM disk, to prevent errors caused by insufficient memory after the swap is disabled. This task is executed by the `crfs_21_install_clearcrfs` script shown in Listing 4.10. First, the actual cleaning script `clearcrfs` is copied to the composite disk image (Line 6). The script is added to shutdown and reboot run levels using `insserv` (Line 16). To make this work, the `clearcrfs` script is declared as a *Boot Facility* in the *System Facility* `local_fs` (Line 12) in the configuration file of `insserv`. This ensures that the `clearcrfs` is executed during a shutdown or reboot together with the other scripts that unmount the local file systems.

```
5   # copy the clearcrfs scripts to the composite root file system
6   cp /scripts/crfs.d/clearcrfs ${ROOT}/etc/init.d
7
8   # patch the insserv configuration, so clearcrfs is put in the local_fs group
9   INSSERVCONF=${ROOT}/etc/insserv.conf
10  if ! grep -qs '^$local.*clearcrfs' ${INSSERVCONF}
11  then
12    sed -i -e '/$local_fs/ s/$/ +clearcrfs/' ${INSSERVCONF}
13  fi
14
15  # register clearcrfs
16  chroot /root insserv clearcrfs
```

Listing 4.10 Script Installing the clearcrfs Init Script. This script named `crfs_21_install_clearcrfs` installs the init script shown in Listing 4.11, by copying it to the composite disk image and adding it to the set of init scripts that are executed during shutdown and reboot.

The actual `clearcrfs` script is shown in Listing 4.11. Together with the Boot Facility declaration, the Linux Standard Base (LSB) [89] (Lines 2 to 10) ensures that the script is executed before the `umountfs` script (Line 5) and both for a shutdown and a reboot, as indicated by the runlevels 0 and 6 (Line 7).

The script first determines the device node of the writable layer (Line 15) and does nothing if the writable layer is not a RAM disk. The actual cleaning procedure takes place in Lines 22 to 24. It consists of two steps. In the first steps, everything except

```
#!/bin/sh
### BEGIN INIT INFO
# Provides:          clearcrfs
# Required-Start:
# Required-Stop:     umountfs
# Default-Start:
# Default-Stop:      0 6
# Short-Description: Clear a tmpfs based writable crfs layer.
# Description:
### END INIT INFO

# determine the type of the writable layer
DEVICE=$(grep -F "/root/.crfs_wl" /proc/mounts | cut -d ' ' -f 1)

# skip the unmount for tmpfs writeable layers, directly halt the system
if [ "${DEVICE}" = "tmpfs" ]
then
  log_action_begin_msg "Cleaning up crfs tmpfs layer"
  # remove everything except /etc, /root (cleared below) and whiteout dirs
  find /root/.crfs_wl -mindepth 1 -maxdepth 1 -type d ! -name etc ! -name root ! -name \.wh\* -exec rm -rf {} \;
  # remove everything below the mountpoint
  find /root/.crfs_wl/root -mindepth 1 -maxdepth 1 ! -name .crfs_wl -exec rm -rf {} \;
  log_action_end_msg $?
fi
```

Listing 4.11 Init Script Clearing a RAM Disk Based Writable Layer. This script named `clearcrfs` deletes everything from a RAM disk used as writable layer that is not required. It keeps the `/etc` directory (containing the script itself), the mount point of the writable layer, and directories used internally by aufs.

the `/etc` and `/root` directories as well as some internal aufs files and directories is deleted from the RAM disk. The `/etc` directory is kept, because it contains important configuration files, such as the modified init scripts itself. The `/root` directory is cleaned in the second step by deleting everything except the mount point of the writable layer.

A solution for the second issue is more complicated. To prevent corruption of the writable layer, remounting it read-only like the `umountroot` script does with a regular root file system seems natural. The `umountfs` script obviously cannot unmount the writable layer, because it is still part of root file system at the time the script is executed. A script inserted between `umountroot` and `halt` scripts should be able to remount the writable layer. Unfortunately, the kernel refuses to do so:

```
# cat /proc/mounts
(...)
/dev/vda1 /.crfslayer1 ext3 ro,relatime,errors=continue,(...)
/dev/vdb1 /root/.crfs_wl ext3 rw,relatime,errors=continue,(...)
aufs / aufs rw,relatime,si=f2f04c6a999db2 0 0
(...)
# mount -o remount,ro /dev/vdb1
mount: /root/.crfs_wl is busy
```

Apparently, it is not possible to remount a layer that is part of an active composite disk image[11]. To successfully remount or unmount the writable layer, a way to unmount the root file system needs to be found. As described above, this is only possible when no process refers to it as its root file system. The modified shutdown procedure shown in Figure 4.17 achieves this. The reboot sequence is almost identical, except that the `rc` script executes the init scripts in `/etc/rc6.d` and `reboot` script is called as final step instead of the `halt` script.

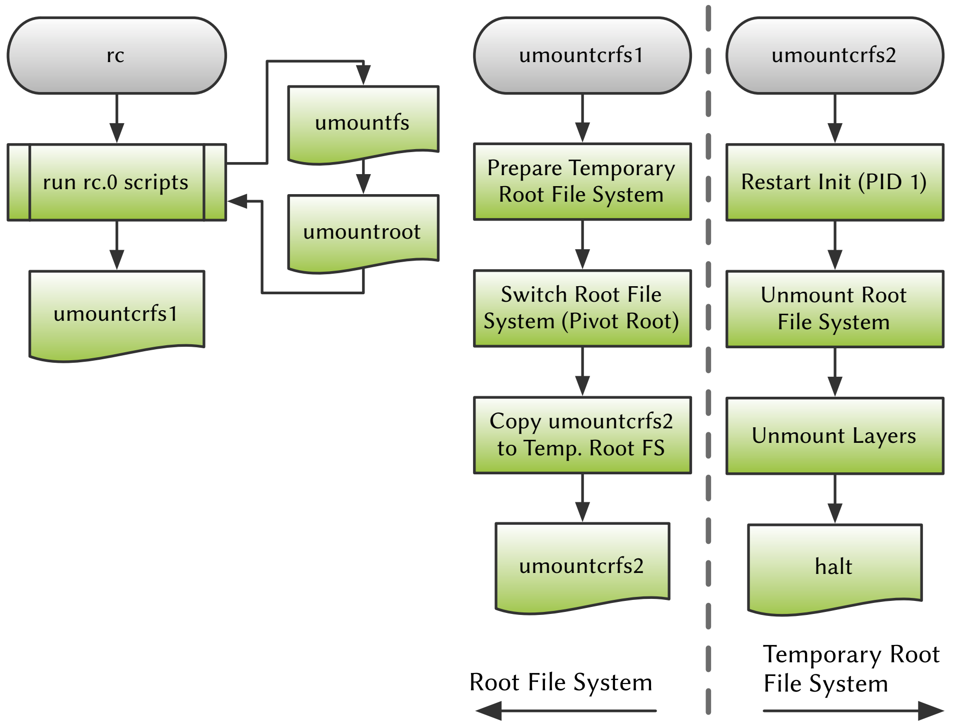


Figure 4.17 The Modified Shutdown Procedure for a Composite Root Image. This flow chart shows the lasts steps in the modified shutdown procedure that handles a composite disk image correctly: the `rc` script executes all scripts in `/etc/rc0.d/` except `halt`, and afterwards executes the `umountcrfs1` script that creates a temporary root file system, switches to it and executes `umountcrfs2` from it. The `umountcrfs2` script in turn restarts init, unmounts the actual root file system and finally halts the system.

The procedure is based on the idea of switching to a temporary root file system that is created dynamically and used solely to unmount the actual root file system. The initramfs used during the boot process (see Phase 5 in Section 4.3.3.1) would be an ideal choice, but is not available in all scenarios, e.g., in Xen PV virtual machines with an external initramfs. The base layer, on the other hand, is always available and can be used as temporary root file system. To preserve its integrity, it has to be mounted read-only and combined with a RAM disk to create a writable temporary root file system.

The script `crfs_22_install_umountcrfs` shown in Listing 4.12 implements the

[11] At least for an implementation based on aufs.

necessary changes in the init system. It first copies the two unmount scripts to the composite disk image (Line 6). Then, it disables the `halt` and `reboot` scripts (Line 9), so the control flow returns to the `rc` script. Figure 4.18a depicts the mounted file systems at this stage of the shutdown procedure for an exemplary composite disk image consisting of two layers. Note that no mount point exists for the lower, read-only layer of the composite disk image. There are only two remaining processes at this point: the shell executing the `rc` script and the PID 1 process of the init system (see Figure 4.19a). The last step is to modify the `rc` script to make it transfer control to the first unmount script (Lines 16 to 18). It is important not to create a new process, but to reuse the existing shell that is executing the `rc` script utilizing the `exec` built-in command of the shell. This ensures that there will still be only the two processes mentioned above. Any additional process will break the shutdown procedure. Thus, it is not possible to just add a standard init script that replaces `halt` or `reboot` instead of modifying the `rc` script.

```
5   # copy the umountcrfs scripts to the composite root file system
6   cp /scripts/crfs.d/umountcrfs? ${ROOT}/etc/init.d
7
8   # disable halt and reboot, register umountcrfs1 instead
9   chroot /root insserv -r halt, reboot
10
11  # patch /etc/init.d/rc to call umountcrfs after all init scripts are finished
12  # for the runlevels 0 and 6
13  RC=${ROOT}/etc/init.d/rc
14  if ! grep -qs 'exec /bin/bash /etc/init.d/umountcrfs1 \$runlevel$' ${RC}
15  then
16    sed -i -e '/^exit 0/ i \
17  #patched by crfs\
18  [ "$runlevel" = "0" ] || [ "$runlevel" = "6" ] && exec /bin/bash
        /etc/init.d/umountcrfs1 $runlevel' ${RC}
19  fi
```

Listing 4.12 Script Patching the Shutdown Procedure. This script named `crfs_22_install_umountcrfs` modifies the `/etc/init.d/rc` script that controls the switching between runlevels. With the modification, it transfers control to `umountcrfs1` (Listings 4.13 and 4.14) during shutdown or reboot, instead of just exiting. Furthermore it disables the `halt` and `reboot` init scripts that would halt or reboot the system before the modified part of `rc` would be reached.

The first unmount script `umountcrfs1` is shown in Listings 4.13 and 4.14. As a first step, the script determines the device node of the writable layer (Line 11) and does just halts or reboots the system if the writable layer is a RAM disk (Lines 14 to 23, not shown). The remainder of the code shown in Listing 4.13 prepares the temporary root file system. In Line 26 the device node of the lowest layer is determined. This works because all layers are still listed in `/proc/mounts`, although they are no longer accessible.

The temporary root file system consists of the lower, read-only layer and a writable RAM disk (*RAM Disk 1* in Figure 4.18) that are combined using aufs[12] exactly like the composite disk image. To prepare it, an additional RAM disk (*RAM Disk 0* in Figure 4.18)

[12] Or the equivalent union mount implementation.

```
 7  # the mountpoint of the writable layer
 8  MOUNTPOINT=/root/.crfs_wl
 9
10  # determine the writable layer device
11  DEVICE=$(grep -F $MOUNTPOINT /proc/mounts | cut -d ' ' -f 1)
12
13  # skip the unmount for tmpfs writable layers, directly halt the system
14  if [ "${DEVICE}" = "tmpfs" ]
 ⋮
23  fi
24
25  # determine the device of the lowest layer (the base layer)
26  DEVICE=$(grep -F /.crfslayer1 /proc/mounts | cut -d' ' -f1)
27
28  echo 'umountcrfs1: creating new temporary root'
29  # container for mounts (root filesystem is ro)
30  mount -nt tmpfs tmpfs mnt
31  # create mountpoints
32  mkdir mnt/ro mnt/rw mnt/root
33  # mount rw layer (tmpfs)
34  mount -nt tmpfs tmpfs mnt/rw
35  # mount the base layer
36  mount -no ro ${DEVICE} mnt/ro
37  # create aufs
38  mount -nt aufs -o br=mnt/rw=rw:mnt/ro=ro aufs mnt/root
39  # create mountpoint for aufs root filesystem
40  mkdir /mnt/root/old
```

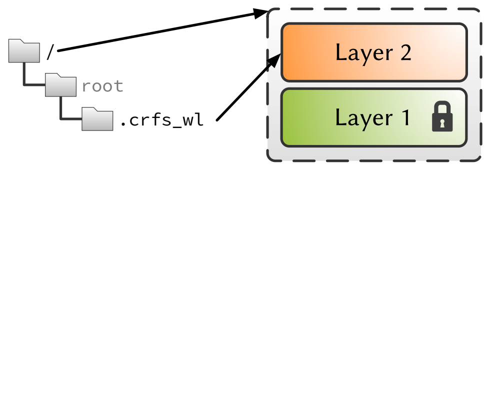
Listing 4.13 First Unmount Script for the Root File System (Part 1). This part of the script named `umountcrfs1` is preparing the temporary root file system.

needs to be mounted at an existing mount point, since the root file system is already remounted read-only (Line 30). The `/mnt` directory has been chosen, because it is part of the *Filesystem Hierarchy Standard* (FHS) [47] that is mandatory for any Linux distribution compliant to LSB. Three mount points are created below `/mnt` (Line 32), the lower layer and the RAM disk (*RAM Disk 1*) are mounted (Lines 34 and 36), and a union mount of the latter is created (Line 38). Figure 4.18b depicts the mounted file systems and Figure 4.19b depicts the running processes and their corresponding root file systems at this stage of the shutdown procedure. Finally, a mount point for the root file system is created inside the temporary root file system (Line 40).

In the next step, the root file system of the current process – the shell that executes `umountcrfs1` – is changed to the temporary root file system using `pivot_root` (Line 44 in Listing 4.14). Then, the remaining system directories and the mount points of the additional RAM disk (*RAM Disk 0*) and the writable layer are moved to the temporary root file system (Lines 46 to 54). Figure 4.18c depicts the mounted file systems after this step[13].

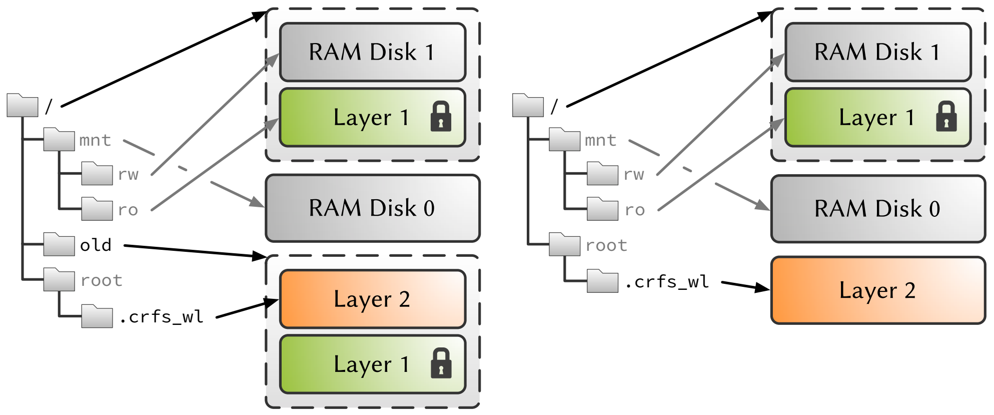
Finally, the second unmount script `umountcrfs2` is copied to the temporary root file system and executed in a newly created shell (Lines 58 and 60). The desired runlevel is passed as an argument. There are two vital requirements to this step: the newly created shell has to be started from the temporary root file system, so that it does not

[13] Note that the positions of the two union mounts are swapped for better presentability.



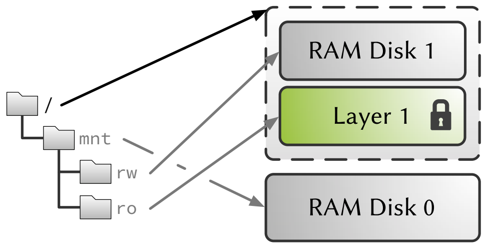
a) At the end of the `rc` script.

b) After the temporary root file system has been prepared in the script `umountcrfs1`.



c) After executing `pivot_root` and moving the mount points in the script `umountcrfs1`.

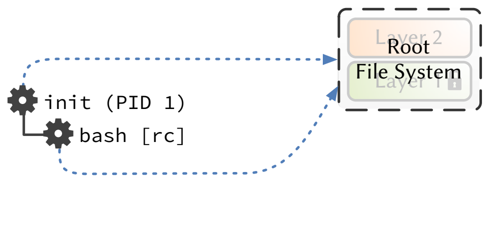
d) After unmounting the root file system in the script `umountcrfs2`.



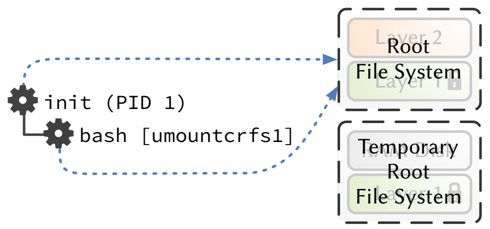
e) After unmounting the writable layer in the script `umountcrfs2`.

Figure 4.18 Mounted File Systems during Shutdown. An overview of the file systems mounted at certain points of the modified shutdown procedure. Less important mounts are depicted in gray. In this example, a two-layer configuration consisting of a read-only base layer *Layer 1* and a writable user layer *Layer 2* is used. The changes done to the user layer should be preserved.

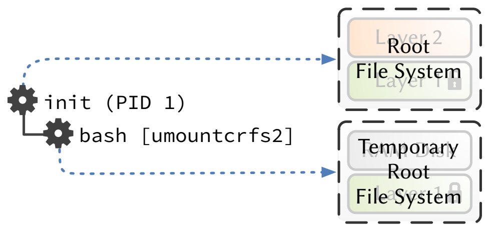
reference the real root file system, and no new process may be created, but the existing one must be replaced. The former requirement is already satisfied by changing the



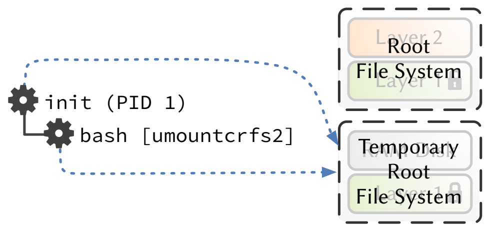
a) At the end of the rc script.



b) Before executing pivot_root in the script umountcrfs1.



c) At the beginning of the script umountcrfs2.



d) Before unmounting the real root file system in the script umountcrfs2.

Figure 4.19 Processes and their Respective Root File Systems. An overview of the remaining processes and their respective root file systems at certain points of the modified shutdown procedure. The same layer configuration as in Figure 4.18 is used.

root file system of the current process, whereas the latter will be satisfied by using the `exec` built-in command of the shell. If done correctly, after this step there are still only two processes, and only one of them still references the root file system: the PID 1 process of the init system. This is depicted in Figure 4.19c.

The remaining reference to the root file system can be cleared by restarting init using the `telinit` command, and thus forcing it to reload itself from the temporary root file system (Line 9 in Listing 4.15). Afterwards, no process referencing the root file system is left (Figure 4.19d), so the root file system can be unmounted in Lines 14 to 24. It is important to use `umount.aufs` for this step (Line 19), so pseudo-links, i.e., hard links over branches, can be persisted. This requires the root file system to be in writable mode, so it is remounted again (Line 18). Figure 4.18d depicts the mounted file systems at this stage of the shutdown procedure. The writable layer can now be unmounted as usual to preserve its integrity (Line 26), whereas the read-only layers of the composite disk image cannot be unmounted, because they are no longer accessible. This is not a problem, because they are guaranteed to be in pristine state because they where not mounted writable. The remaining mounted file systems are depicted in Figure 4.18e. As the only file system that is not a RAM disk is mounted read-only, the system can be halted or rebooted at this point (Lines 29 to 34).

```
echo 'umountcrfs1: switching into temporary root filesystem'
# pivot into temporary root filesystem
sbin/pivot_root /mnt/root /mnt/root/old
# move system mounts to temporary root filesystem
mount --move /old/proc proc
mount --move /old/sys sys
mount --move /old/dev dev
mount --move /old/lib/init/rw /lib/init/rw
# move container for mounts to temporary root filesystem
mount --move /old/mnt mnt
# move the writable layer mountpoint to temporary root filesystem
mkdir ${MOUNTPOINT}
mount --move /old${MOUNTPOINT} ${MOUNTPOINT}

echo 'umountcrfs1: switching to second script'
# copy second script to temporary root filesystem
cp /old/etc/init.d/umountcrfs2 /etc/init.d/umountcrfs2
# exec the script in a new bash, loaded from the temporary root
exec /bin/bash /etc/init.d/umountcrfs2 ${RUNLEVEL}
```

Listing 4.14 First Unmount Script for the Root File System (Part 2). This part of the script named `umountcrfs1` is changing the root directory to the temporary root file system and executes `umountcrfs2` (Listing 4.15) using the new root directory.

4.5 Experimental Results

This section contains an evaluation of composite disk images both with respect to storage efficiency and to runtime impact. Finally, the benefits of using composite disk images in Virtualized Grid Computing environments are evaluated based on a concrete use case.

Unless stated otherwise, all measurements have been conducted on MaRC, the former compute cluster of the University of Marburg. Each of the nodes in the cluster contained two Dual-Core AMD Opteron 2216HE CPUs running at 2.4 GHz, 16 GiB RAM, and a 250 GB SATA hard disk. The nodes were interconnected using a 1 Gbit switched Ethernet network. At the time of the measurements, Debian GNU/Linux 4 (Etch) was used as operating system and Xen 3.0.2 was used as hypervisor.

4.5.1 Storage Efficiency of Composite Disk Images

To measure the storage efficiency of composite disk images, a set of 31 Debian GNU/Linux 6 virtual machines have been created. In the remainder of this thesis, this set is called Set *A*. It will also be used in the evaluation of the Marvin Image Store (see Section 5.5). An overview of Set *A* is given in Table 4.8.

The first virtual machine (*A01*) contains a base installation of Debian GNU/Linux 6 and a few additional packages. This virtual machine was cloned thirty times and each of the clones (*A02* – *A31*) was modified by installing an additional piece of software and all its dependencies. The *Description* column of Table 4.8 contains the name of the

```
echo 'umountcrfs2: restarting init'
# restart init, load it from the temporary root
telinit u

echo 'umountcrfs2: umount composite root filesytem'
# unmount the composite root filesytem and the writable layer
# check for the existence of umount.aufs (aufs-tools)
if [ -x /sbin/umount.aufs ]
then
  # remount old root as writable, so auplink can persist
  # the plinks (auplink is invoked by umount.aufs)
  mount -o remount, rw /old
  /sbin/umount.aufs /old
else
  # warn user and do a simple umount
  echo 'umountcrfs2: WARNING: auplink (aufs-tools) not installed. can
    not cleanly unmount writable layer.'
  umount /old
fi
# unmount the writable layer
umount /root/.crfs_wl

echo 'umountcrfs2: done'
if [ ${RUNLEVEL} -eq 0 ]
then
  /etc/init.d/halt stop
else
  /etc/init.d/reboot stop
fi
```

Listing 4.15 Second Unmount Script for the Root File System. This script named `umountcrfs2` restarts init to cut the last tie to the old root directory – the composite disk image, unmounts both the composite disk image and the writable layer and shuts down or reboots the system.

added software. In most cases (23), all packages containing the name of the software to install were installed using a command like the one below for *Octave*:

```
export DEBIAN_FRONTEND=noninteractive
aptitude search 'octave' | awk '{print $2}' | xargs apt-get install -y
```

Installation recipe for the virtual machine containing the Octave installation.

In the remaining cases, manually adjusted installation commands were used, either to remove specific packages causing problems from the list returned by `aptitude search` using `grep`, or by manually specifying the list of packages to install when automatic list generation failed. The generation of the 31 virtual machines was automated using a Ruby program that relies on recipe files with the mentioned installation commands to prepare the virtual machines. This facilitates the easy creation of a large number of virtual machines with contents ranging from web applications over services to software development environments.

In addition to regular virtual machine images, the virtual machines *A02* – *A31* where also created a second time using a composite disk image. The disk image of the first

Table 4.8 Virtual Machines of Set *A* – Content. This table contains a description and some statistics about the virtual machines of Set *A*. All sizes are measured from inside the virtual machines, are thus restricted to the files itself and are ignoring the file system overhead.

		Number of Files		Content Size (MiB)		
ID	Description	Regular	Layered	Regular	Layered	Savings
A01	base	17,432	—	474.7	—	—
A02	ejabberd	18,843	1,485	504.0	47.3	90.6 %
A03	roundup	19,157	1,799	505.7	49.2	90.3 %
A04	afpfileserver	24,716	7,362	539.8	83.7	84.5 %
A05	mantis	19,935	2,575	555.1	98.5	82.3 %
A06	movabletype	23,948	6,595	580.6	124.3	78.6 %
A07	wordpress	20,727	3,310	583.6	129.2	77.9 %
A08	redmine	24,663	7,295	589.7	133.2	77.4 %
A09	moinmoin	23,450	6,098	597.0	140.7	76.4 %
A10	drupal	20,694	3,341	613.0	156.6	74.5 %
A11	lapp	24,113	6,761	621.9	165.8	73.3 %
A12	dokuwiki	24,913	7,561	626.4	170.6	72.8 %
A13	phpbb	26,579	9,227	647.2	190.9	70.5 %
A14	lamp	23,178	5,827	655.2	199.0	69.6 %
A15	moodle	28,213	10,863	682.5	226.2	66.9 %
A16	otrs2	24,578	7,226	693.0	236.8	65.8 %
A17	mediawiki	25,003	7,654	700.6	244.4	65.1 %
A18	openjdk	21,452	5,896	705.5	256.0	63.7 %
A19	mono	22,197	4,843	707.2	251.3	64.5 %
A20	django	29,715	12,362	722.2	266.4	63.1 %
A21	yorick	30,366	13,012	737.1	281.3	61.8 %
A22	typo3	27,746	10,395	743.6	287.8	61.3 %
A23	gallery	27,076	9,739	765.2	318.4	58.4 %
A24	tomcat	23,622	8,068	797.1	347.6	56.4 %
A25	pootle	30,328	12,977	798.5	342.7	57.1 %
A26	ruby19	43,807	26,454	933.7	477.9	48.8 %
A27	gforge	38,253	20,905	982.5	526.4	46.4 %
A28	trac	36,191	18,855	1,015.2	568.3	44.0 %
A29	r	40,508	24,958	1,200.1	750.8	37.4 %
A30	octave	47,858	30,511	1,469.5	1,013.9	31.0 %
A31	haskel	27,686	10,342	1,635.4	1,181.8	27.7 %

virtual machine (*A01*) was used as base layer. Table 4.8 shows the number of files in all virtual machine images and their total size, both for layered and regular virtual machines. The layers are between 446.8 MiB and 456.7 MiB (454.7 MiB on the average) smaller compared to the regular versions of the virtual machines. When considering the size of the base layer, this means that 95.8 % of the contents of the base layer are reused. Thus, the overall savings by using composite disk images range from 27.7 % to 90.6 %, depending on the size of additional software installed in the layer, with an average saving of 64.6 %.

The size of all files in the virtual machine image is important, as there is not necessarily a direct relation to the size of the corresponding disk image. This can be seen in

Table 4.9 that shows the size of the disk images of all virtual machines. The discrepancy of up to 723.5 MB between the content size and the image size (*A24*) is caused by the installation procedure, during which a lot of temporary files are written to disk and deleted afterwards. All regular virtual machine images use 4 GiB sparse disk images, whereas the layers are stored in 2 GiB sparse disk images. Sparse disk images only occupy as many sectors on the host's disk as are used. The actual size of a 2 GiB sparse disk image can thus range between a few MiB[14] and 2 GiB.

Table 4.9 Virtual Machines of Set *A* – Image Sizes. This table contains disk image size information for all virtual machines of Set *A*: both for optimized and non-optimized disk images the sizes of regular and layered virtual machine disk images are compared.

	Image Size (MiB)			Optimized Image Size (MiB)		
ID	Regular	Layered	Savings	Regular	Layered	Savings
A01	1,190.1	—	—	668.2	—	—
A02	1,214.5	161.9	86.7 %	697.3	144.1	79.3 %
A03	1,222.9	152.2	87.6 %	699.1	146.0	79.1 %
A04	1,249.3	202.2	83.8 %	733.2	180.5	75.4 %
A05	1,278.6	231.6	81.9 %	748.5	195.3	73.9 %
A06	1,301.1	279.7	78.5 %	774.0	221.1	71.4 %
A07	1,290.7	271.8	78.9 %	777.0	226.0	70.9 %
A08	1,298.0	264.3	79.6 %	783.2	230.0	70.6 %
A09	1,303.3	285.3	78.1 %	790.4	237.5	69.9 %
A10	1,313.2	316.2	75.9 %	806.4	253.4	68.6 %
A11	1,302.7	315.7	75.8 %	815.3	262.6	67.8 %
A12	1,325.3	328.3	75.2 %	819.8	267.4	67.4 %
A13	1,356.0	360.1	73.4 %	840.6	287.6	65.8 %
A14	1,341.9	359.3	73.2 %	848.6	295.7	65.1 %
A15	1,355.1	390.9	71.2 %	875.9	323.0	63.1 %
A16	1,363.1	415.8	69.5 %	886.4	333.6	62.4 %
A17	1,382.7	414.6	70.0 %	894.0	341.2	61.8 %
A18	1,423.0	468.3	67.1 %	898.9	352.8	60.8 %
A19	1,413.8	433.2	69.4 %	900.5	348.1	61.3 %
A20	1,443.3	439.6	69.5 %	915.6	363.1	60.3 %
A21	1,445.2	468.3	67.6 %	930.5	378.1	59.4 %
A22	1,423.7	498.8	65.0 %	936.9	384.6	59.0 %
A23	1,471.5	528.9	64.1 %	958.6	415.2	56.7 %
A24	1,505.0	575.6	61.8 %	990.5	444.3	55.1 %
A25	1,412.6	511.6	63.8 %	991.8	439.4	55.7 %
A26	1,634.7	717.2	56.1 %	1,127.1	574.6	49.0 %
A27	1,581.7	765.6	51.6 %	1,175.8	623.1	47.0 %
A28	1,715.7	859.4	49.9 %	1,208.5	665.0	45.0 %
A29	1,904.5	1,169.2	38.6 %	1,393.4	847.4	39.2 %
A30	2,160.7	1,448.5	33.0 %	1,662.6	1,110.5	33.2 %
A31	2,344.6	1,491.4	36.4 %	1,828.8	1,278.6	30.1 %

Simply by copying their contents to a blank disk image, the size of a disk image can

[14] The minimal size depends on the file system inside the disk image. Before a file system is created its size could even be 0 MiB.

be reduced significantly. The image size is reduced between 22.0 % and 43.9 % (on the average 35.6 %) for regular images and between 4.1 % and 27.5 % (on the average 18.2 %) for layered images. The effects of this optimization are questionable for regular virtual machine images, because normal use of the virtual machine will undo the optimization. For composite disk images on the other hand, the optimization is reasonable, because the layers will not be modified during usage.

Table 4.9 contains the sizes of both regular and layered images, with and without optimization, as well as the savings resulting from the use of composite disk images. In the non-optimized case, the savings range from 33.0 % to 87.6 % with an average of 67.8 %. While the savings with regard to the image size are lower for the smaller virtual machines (*A02* – *A06*) compared to the content size, for the other virtual machines the savings are even higher. The reason is that the overhead of a second file system in the layer containing the additional software is dominant for smaller layers, but negligible for bigger layers. Obviously, the savings are reduced by 7.0 % on the average when optimization is used for both regular and layered virtual machines.

4.5.2 Runtime Impact of Composite Disk Images

The use of redirect-on-write technology in image composition introduces another layer of abstraction to I/O operations. Especially virtual machines performing I/O intensive tasks might suffer from a performance degradation caused by their use of composite disk images. Two measurements have been conducted to evaluate the overhead of image composition: a synthetic benchmark and a compilation of the Linux kernel.

4.5.2.1 bonnie++ Benchmark

The *bonnie++* [29] benchmark, a well-known testing suite aimed to perform a number of file system related tests, has been used as synthetic benchmark. A Debian GNU/Linux 4 virtual machine with a single CPU core and 128 MiB RAM was used for this test[15]. The virtual machine image was used in two scenarios: as a regular virtual machine and as a layered virtual machine, by combining its disk image with an empty, writable layer. All disk images were allocated completely before the tests to prevent any bias caused by delayed allocation when sparse files are used. A total of 100 tests was performed each of the scenarios, using a 256 MiB file for the I/O performance measurement. There are 6 file I/O tests in bonnie++ that are briefly described in Table 4.10.

For the random seek test, bonnie++ could not calculate accurate results, because the test completed too fast and rounding errors would probably bias the results [29]. Therefore, the evaluation focuses on the other 5 tests, whose results are shown in Figure 4.20.

There are only small differences in the throughput values measured by bonnie++ between the regular and layered virtual machine. In both the write block and write

[15] In Debian GNU/Linux 4 aufs was not available, instead UnionFS was used for image composition.

Table 4.10 File I/O Tests of bonnie++. A description of the tests in bonnie++ that focus on file I/O ([29]).

Test	Description
Block Write	A file is written using `write()` system call, writing the data in blocks.
Block Read	A file is read using `read()` system call, reading the data in blocks.
Character Write	A file is written using the `putc()` macro of the C standard library, writing the data in individual characters.
Character Read	A file is written using the `getc()` macro of the C standard library, reading the data in individual characters.
Rewrite	This is a combined test. The data is first read from the file using `read()`, modified in memory, and then written back using `write()`. Before writing, the offset in the file is changed using the `lseek()` system call.
Random Seek	Multiple processes are executing 8000 random `lseek()` system calls, reading a block using `read()`, and in 10 % of the cases modifying and writing the block back using `write()`.

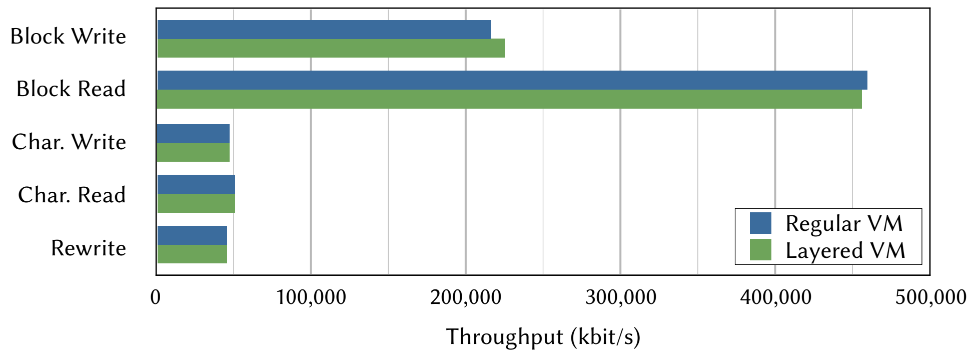


Figure 4.20 Results of the bonnie++ Benchmark. The throughput measured by bonnie++ in the first five tests described in Table 4.10, both for the regular and layered virtual machine.

character tests, the layered virtual machine outperforms the regular virtual machine by 3.8 % and 1.8 %, respectively. In the remaining three tests, the regular virtual machine outperforms the layered virtual machine by 0.4 % in the read block and read character tests and 1.0 % in the rewrite test. The latter results were expected because of the complexity added by the use of a union mount. The results in both the write block and write character tests, on the other hand, were unexpected and the reason why the layered virtual machine was able to outperform the regular virtual machine is not clear. The only reasonable explanation is an odd interaction with the disk cache in the dom0 that might be able to cache a larger amount of write accesses for the empty layer than for the regular disk image containing the operating system. Additionally, the disk cache is also very likely the cause of the high I/O throughput values for block read and block write that are way beyond the capability of a single SATA hard disk, suggesting that large portions of the read and write accesses were cached.

4.5.2.2 Linux Kernel Compilation

The second measurement was the compilation time of the Linux kernel inside a virtual machine. It has been chosen because it represents a balanced workload stressing the virtual memory system, doing moderate disk I/O as well as being relatively CPU intensive. In contrast to the other measurements in this chapter, this measurement has been done on a workstation with an Intel Pentium 4 651 CPU running at 3.4 GHz, 1 GiB RAM and a 250 GB SATA hard disk. Debian GNU/Linux 5 (Lenny) was used as operating system and Xen 3.2.1 was used as hypervisor.

A virtual machine with two virtual CPUs[16] and 128 MiB was used for this test in three different configurations: as regular virtual machine and as a virtual machine with two or three layers. For both layered virtual machines, the lowest layer was a copy of the disk image of the regular virtual machine, i.e., a complete Debian GNU/Linux 5 installation including all required software for compiling the kernel. The kernel sources itself were extracted in the second lowest layer. In case of the virtual machine with three layers, the highest layer was initially empty at the start of the virtual machine.

The default configuration of the Debian GNU/Linux 5 kernel was used for building the Linux 2.6.18 kernel. The command used to compile the kernel was `make -j 4`, to ensure that both of the virtual CPUs are used by allowing four simultaneous jobs. In this specific case, a job is any command that is executed by make to build the kernel, e.g., the C compiler, the assembler, or the linker. Although the virtual machine has only two virtual CPUs, a maximum number of 4 simultaneous jobs is used, because a factor between 1.5x and 2.0x is generally recommended when calculating the maximum number of jobs. The Linux kernel has been compiled ten times in each of the three configurations.

A comparison of the elapsed (wall clock) time for the compilation is shown in Figure 4.21. The runtime overhead caused by using image composition with two or three layers is 3.0 % or 3.6 %, respectively. This value is a little bit higher than the result of the bonnie++ benchmark, although still acceptable.

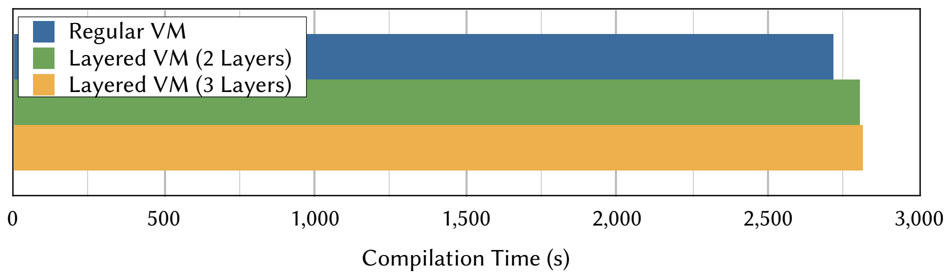


Figure 4.21 Linux Kernel Compilation – Elapsed Time. This chart shows a comparison of the elapsed (wall clock) time for compiling the Linux kernel in either a regular virtual machine or a layered virtual machine with two or three layers, respectively.

[16] The Pentium 4 651 CPU supports Hyper-Threading, Intel's implementation of hardware multithreading. It consists of two logical processors that share execution resources.

More detailed metrics gathered by GNU time can be found in Table 4.11. These values show that the number of file system input operations increased by 18.9 % or 28.2 % for the layered virtual machines using two or three layers, respectively. The reason for this increase is the need to read metadata from two or three file systems in order to use the union mount. This is likely the explanation of the higher overhead compared to the bonnie++ benchmark. Another potential reason for the increase could be copy-up operations, i.e., copying file to the writable layer before they are modified. But in this specific case the number of modified files is relatively low, because it mostly consists of reading source files and writing object files. The number of output operations, on the other hand, does almost not increase at all, because all write operations are executed on the highest layer and all other layers are read-only.

Table 4.11 Linux Kernel Compilation – Detailed Results. This table lists average values of a few metrics reported by GNU time that can help to assess the overhead of using image composition.

		Layered VMs	
Measurement	Regular VM	2 Layers	3 Layers
Elapsed (Wall Clock) Time (s)	2,718.0	2,800.4	2,815.0
User Time (s)	4,869.4	4,831.9	4,836.1
System Time (s)	523.0	630.9	650.9
Major Page Faults	10,745.7	14,081.0	15,888.2
File System Inputs	1,135,396.8	1,349,304.0	1,455,352.0
File System Outputs	956,396.0	956,524.0	956,583.2

The higher overhead is also directly related to the increased *system time*, i.e., time spent in the kernel. This increase is caused by the union mount that is implemented as a file system driver in the kernel. Strangely, the *user time* decreases in the layered virtual machines. There is no obvious explanation for this effect. The higher amount of *page faults* also hints at a memory shortage caused by the need to cache more file system metadata. With more memory, the overhead caused by using image composition might be less.

4.5.2.3 Discussion

The measurements show that using a composite disk image as root file system does not have a big impact on the I/O throughput. Additionally, the amount of I/O on the composite disk image should be rather limited in most use cases, because the composite disk image intentionally offers no persistent storage to enable the reuse of disk images. Most of the data read and written during the runtime of a virtual machine using a composite disk image is likely located on an external storage system, mounted via NFS.

4.5.3 Virtualized Grid Computing Use Case

The last part of the evaluation is Virtualized Grid Computing scenario. Consider a user that prepares a virtual machine and installs all software that is required to execute the specific kind of job the user needs to compute in the Grid. When the scheduler grants an execution slot to the user, the virtual machine has to be transferred to one or more compute nodes that are located either at the local Grid site of the user or at a remote Grid site. Each time the user submits a new job to be executed in this virtual machine, it has to be transferred again.

In most cases, virtual machines can be reused, i.e., the same virtual machine that has been used to execute the first job of a user might also be used to execute another job of the same type[17]. Thus, the compute nodes might keep virtual machines images in a local image cache after use. The drawback of this approach is the limited amount of local storage typically available to compute nodes (250 GiB in the case of MaRC), so the number of virtual machines that can be cached is limited. Additionally, this local storage might also be required as scratch space for jobs to keep the load on the NFS shares lower. A small cache combined with a large number of virtual machines, the typical size of virtual machines, and a scheduler that does not take contents of those caches into consideration when assigning a job to a compute node probably renders the use of virtual machine caching useless.

If the user prepares his virtual machine using a composite disk image and a common base layer, caching becomes useful again despite all the drawbacks described above. The reason is the common base layer that is not only used by a single virtual machine, but by a large number of virtual machines. So, instead of caching complete virtual machine images, the compute nodes only cache a single base layer. Even if different virtual machines use different base layers, the cache will be much more effective when composite disk images are used.

The virtual machine used in this scenario contains a base installation of 162 packages using about 468 MiB. The user installs 14 additional packages using about 58 MiB. Both the regular virtual machine and the base layer are stored in a 4 GiB sparse disk image using the ext3 file system, while the user layer is stored in a 1 GiB sparse disk image. The differences between the content size and the size of the image files as shown in Table 4.12 have already been explained in Section 4.5.1.

The transfer times of the three disk images listed in Table 4.12 have been measured in two different situations: between two compute nodes of MaRC, to simulate a local transfer, and between MaRC and the Cluster of the University of Frankfurt via the German Research Network (DFN), to simulate a remote transfer between two Grid sites. Each disk image was transferred 60 times to calculate a robust mean value. Since sparse files are used for the disk images, only the used parts of the image need to be transferred. Unfortunately, this rules out scp, the obvious tool for secure file transmission, because it cannot handle sparse files. Therefore, tar is used over a

[17] Unfortunately, there is no guarantee that an error does not render a virtual machine broken, resulting in errors in subsequent jobs executed in the same virtual machine.

Table 4.12 Size and Transfer Time in a Single and a Multi Site Scenario. Transfer times of a virtual machine, both regular and layered, between compute nodes of a cluster and between two clusters at different Grid sites, optionally compressed.

		Transfer Time (s)		
		Single Site	Multi Site	
Image	Size (MiB)	Uncompressed	Uncompressed	Compressed
Regular Image	691.1	40.6	660.8	460.1
Base Layer	666.4	39.1	636.9	443.5
User Layer	67.7	14.4	101.5	91.6

SSH-encrypted data channel. It is able to handle sparse files and supports compression that is used to speed up the copy operation in the remote transfer case using gzip.

Figure 4.22 shows the amount of data that needs to be transferred to consecutively execute multiple jobs in the virtual machine described above. For the layered virtual machine, four different scenarios have been considered: caching restricted to the base layer(s) and caching of all layers, both with an empty cache and a cache primed with the particular base layer of the virtual machine.

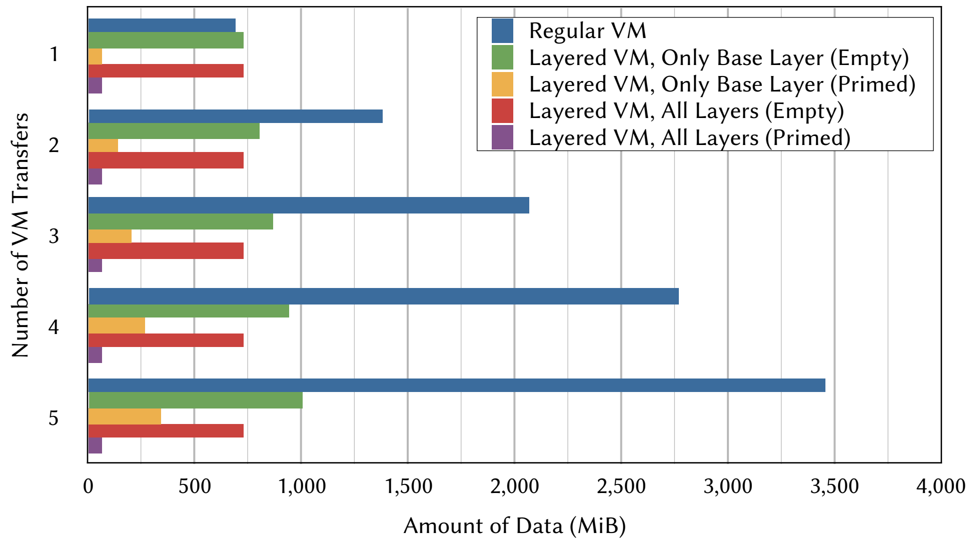


Figure 4.22 Data Transfer to Compute Nodes. Amount of Data that needs to be transferred to a compute node in order to consecutively execute a regular or a layered virtual machine multiple times. For the layered virtual machine, four different caching scenarios are evaluated: caching of base layers and caching of all layers, both for an empty cache and a cache primed with the particular base layer.

There is a slight overhead for the first job if the cache is empty (6.2 %), but starting with the second job the layered virtual machine requires significantly less data to be transferred, even when only the base layer is cached. For five jobs, the amount of data that needs to be transferred reduces by 70.9 % – 98.0 % in the four scenarios. If the job is executed in parallel on multiple compute nodes, which is very likely as the Grid is typically used for compute intensive, parallel jobs, the difference between the necessary data transfer for regular and layered virtual machines might even grow faster, unless special distribution techniques are used.

4.6 Summary

In this chapter, the Marvin Image Compositor, a novel approach for disk image provisioning designed for Linux-based virtual machines was presented. It is based on the idea of composition: a disk image containing a (shared) base installation of an operating system is dynamically combined with another disk image containing the actual application yielding a composite disk image with the combined contents of both disk images (or layers). This approach does not only reduce the size of virtual machine images, but also facilitates reuse and caching of layers because it keeps the layers in pristine state during use.

Several experiments have shown the practicability of the approach. Layers require up to 87 % less disk space than their regular counterparts, while generating a runtime overhead of only 3.6 %. Depending on the use case, the overhead might be even smaller. In Virtualized Grid Environments that might require to copy disk images to execution hosts repeatedly, this approach can save up to 98 % of bandwidth.

"A place for everything and everything in its place."

Proverb dated from the 17th century

5 Virtual Machine Image Storage

5.1 Introduction

The widespread usage of virtual machines in Virtualized Grid and Cloud Computing leads to a massive increase in the number of virtual machines. This phenomenon called *virtual machine sprawl* [128] is a problem for providers both in Virtualized Grid and Cloud Computing, because virtual machines have significant storage requirements: a single virtual machine can require between a few hundred megabytes and a few gigabytes.

The reasons for virtual machine sprawl are manifold. First of all, virtual machines in the Cloud are much cheaper than physical hardware in terms of up-front costs, lessening the barrier of virtual machine creation. The sprawl is increased by the fact that the creation of a virtual machine using one of the virtual machine management systems described in Section 3.2 is literally just a click away. Users nowadays tend to create virtual machines for everything, from testing software to providing isolated environments for individual tasks or pieces of software. While this is a positive development from a security point of view, it poses a challenge for the management of virtual machines.

The sprawl of virtual machines increases even more if multiple versions of each virtual machine are to be preserved. Nevertheless, keeping older versions has many benefits. First and foremost, it provides a safety net for maintenance operations, because installing updates or modifying a virtual machine in other ways can break either the whole virtual machine or a piece of software within. Being able to undo the changes to a virtual machine by switching back to an older, working version makes modifying a virtual machine a less risky operation. Additionally, the possibility to compare different versions of a virtual machine can provide important input for a root cause analysis of problems caused by updates or modifications.

The preservation of older versions of a virtual machine is also a promising concept for computational sciences as it helps to warrant the *reproducibility of results* that is an important requirement in this domain. It facilitates the recreation of the exact environment that has been used to generate specific results, so it can be used at any later time – even by other scientists – to reproduce and thus verify these results.

The traditional form of virtual machine storage – plain disk images – can neither efficiently store large numbers of virtual machines nor many versions of individual virtual machines. Composite disk images based on both shared and individual layers as presented in Chapter 4 are a first step to deal with the storage of large numbers of virtual machines, by applying a coarse grained form of deduplication: shared base layers. More fine-grained deduplication can further reduce the storage requirements of large sets of virtual machines by taking advantage of the similarities between different virtual machine images in a way that is not possible for layers.

In this chapter, a novel approach for virtual machine image storage called *Marvin Image Store* (also referred to as Image Store) is presented that efficiently stores a potentially large number of layered Linux virtual machine images in multiple versions and at the same time provides efficient means to access and update them. It combines the efficient deployment enabled by the use of layer based composite disk images with the advantages of a finer grained deduplication. Furthermore, the Image Store offers advanced operations on stored virtual machines, e.g., comparing virtual machine images, either different versions of the same virtual machine or totally independent virtual machines.

In contrast to virtual machine image composition presented in Chapter 4, which is primarily aimed at optimizing the deployment time of virtual machines and only improves storage efficiency as a side effect, the Image Store presented in this chapter focuses solely on the storage and maintenance phases in the lifetime of a virtual machine. Additionally, the directly related deployment and undeployment phases are affected as well. The phases in the lifecycle of a virtual machine that are related to using the Image Store are shown in Figure 5.1.

Parts of this chapter have been published in [142].

5.2 Related Work

Meyer et al. [100] have conducted a large-scale study on deduplication using desktop machines at Microsoft. Overall, the study consists of 857 file systems spanning 162 terabytes. By using block-based deduplication techniques, the authors achieved storage savings of up to 32%. Using file-based deduplication on backup images (where a new backup image contains a reference to a duplicate file in an old backup), they achieved 87% of the savings of a block-based strategy. Based on their work and own experiments, file-based deduplication was chosen as one of the core technologies of the Marvin Image Store.

Jayaram et al. [72] have presented a comprehensive study about the similarity of virtual machine disk images. The authors analyzed 525 disk images used in a productive

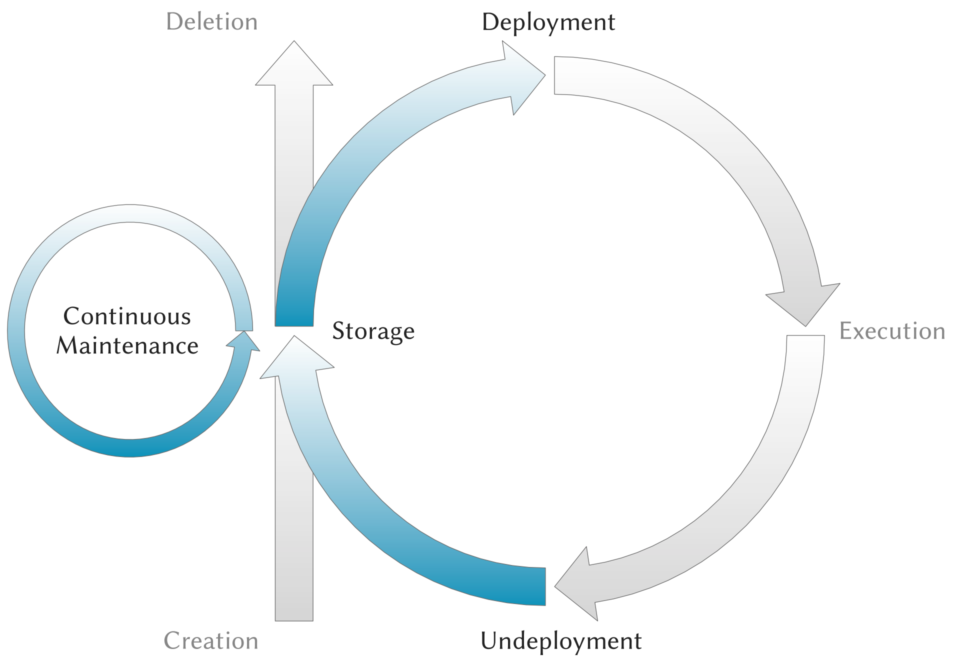


Figure 5.1 Related Virtual Machine Lifecycle Phases. The focus of virtual machine image storage regarding the lifecycle of a virtual machine are the storage and maintenance phases. The directly related deployment and undeployment phases are affected as well.

Cloud computing environment, based on black-box similarity detection techniques. Black-box means that an image is broken into blocks of fixed and variable size. Their results showed that most images contain a significant rate of duplication, and thus quite good compression ratios can be achieved.

Nath et al. [103] have presented a method to effectively store virtual machine disk images. Their approach is based on the assumption that most disk images contain similar content, e.g., the same binaries, configuration and device files. Thus, they use *content-addressable storage* (CAS) to actually reduce the storage requirements. An image is split into fixed size blocks that are hashed using the SHA-1 hash function . Based on the chosen block size and this deduplication strategy, a significant amount of storage can be saved.

A similar approach has been presented by Jin and Miller [74]. The authors further study the effect of different block sizes using the Rabin fingerprint algorithm [124]. According to their results, using a variable block size does not necessarily lead to improved storage savings. Sometimes, variable block sizes could even lead to reduced savings. Therefore, the authors propose using fixed block sizes like. Additionally, they propose compression of blocks, but their evaluation of this idea is only superficial.

Both proposals are comparable to the Marvin Image Store in terms of image storage. Nevertheless, in both approaches a block-based deduplication approach is used in comparison to the file-based approach used by the Marvin Image Store. Although this approach promises higher storage efficiency, small block sizes can vastly increase the manifest size. Neither Nath et al. nor Jin and Miller propose mechanisms to optimize the import and export processes like the Marvin Image Store does.

Experiences with content-addressable storage and disks images have been presented by Liguori and Hensbergen [86]. In contrast to the approaches mentioned above, the authors designed their solution for direct usage of the stored image files by virtual machines, i.e, the disk images do not need to be copied to the execution host before the virtual machine is started. Their implementation is based on the *Venti* [123] CAS system and QEMU. Although their image analysis is very useful, the implementation is only a simple pass-through between QEMU, the 9P file server and Venti and thus not suitable for a Cloud or virtualized Grid environment.

LiveDFS, a kernel-space file system for Linux, is a similar system that has been proposed by Ng et al. [105]. One of the design goals of LiveDFS was reasonable memory consumption to facilitate its use on commodity hardware. This is achieved by keeping only partial metadata in memory and storing the full metadata on disk. Splitting the metadata to reduce the memory consumption has a negative impact on the performance, because of the additional I/O requests required to access the entire metadata. It is compensated by placing the metadata close to the data and using a structure that can be cached efficiently by the Linux disk cache. Like many of the proposals above, LiveDFS uses fixed size blocks for deduplication.

Nicolae et al. [106] have proposed a distributed repository for virtual machine images. The authors advocate aggregating a part of the execution hosts' local disks to build a large distributed storage system. Virtual machine images are split into equally sized chunks and distributed among all disks. Like LiveDFS, the proposed repository is designed for direct usage of the stored image files. There is no indication whether the proposed repository uses deduplication to reduce the storage requirements of virtual machines like the Marvin Image Store does or not.

The proposals of Liguori and Hensbergen, Ng et al, and Nicolae et al. are all designed to be used a storage systems for direct usage. They do not use compression to further reduce the storage requirements of the virtual machine images to provide good access efficiency. Additionally, none of the approaches can provide a deeper insight into the contents of the virtual machine images like the Marvin Image Store does.

Al-Kiswany et al. [6] have proposed *VMFlock*, an approach for efficient deployment of groups of virtual machine images, e.g., a multi-tier application. They use deduplication to reduce the amount of data that needs to be transferred to the execution hosts, both within the group of virtual machine images and between the group and images already available at the execution hosts. VMFlock has been designed as a standalone extension of existing virtual machine repositories, thus it does not consider the on-disk storage requirements of virtual machines.

Both Jeswani et al. [73] and Zhou et al. [178] have proposed systems that improve the caching of virtual machine templates by selecting a base template and storing differences between this base template and other templates. Whenever a template other than the base template is required to instantiate a virtual machine, its template is regenerated by applying the differences to the base template. This approach is comparable to the way the Marvin Image Store updates existing image files.

5.3 Design

An ideal storage system for virtual machines has to deal with a large number of virtual machines, especially when it is deployed in Virtualized Grid and Cloud Computing environments. Therefore it needs to provide high *storage efficiency*, i.e., efficient usage of the available storage to maximize the number of stored virtual machines. This is typically implemented by using deduplication techniques that reduce the storage requirements of virtual machines significantly.

However, storage efficiency is only one of the major challenges faced by a storage system for virtual machines. The other major challenge is *access efficiency*, i.e., efficient access to stored virtual machine images and their contents. This is important because it directly influences the deployment times, which are an important criterion of the perceived performance of a Cloud environment. For some users of Cloud environments fast deployment times are even critical, because they rely on the fast adaption of the Cloud environment for scalability of their applications.

Besides storage and access efficiency, *version control* is a desirable feature of an virtual machine storage system. It should not only store a single disk image per virtual machine, but a disk image for each version of the virtual machine. These disk images have to be stored in an independent manner, so they do not have a negative impact on the access efficiency. The snapshot technology commonly used by hypervisors is an example of a contrary approach: each snapshot only contains the changes with respect to the preceding snapshot and thus depends on it (recursively). To create a copy of the virtual machine at a specific snapshot the underlying disk image needs to be copied and all snapshots up to the desired one need to be applied sequentially to the copy. Thus, this approach is not applicable for a virtual machine storage system.

Another desirable feature of a storage system for virtual machines is *content awareness*, i.e., it should be able to understand the structure and contents of the images it stores. The knowledge of virtual machine image contents facilitates shifting some management and maintenance operations to the storage system itself. This shift of operations to the storage system increases access efficiency, because the image does not need to be fetched and transfered to another host first. Additionally, the knowledge facilitates novel management, maintenance, and analysis operations that are not possible with traditional virtual machine images.

In this chapter, a novel repository for Linux-based virtual machines is presented: the *Marvin Image Store*. Its design is inspired by prior work of Wei et al. [167] and Reimer et al. [128]: *Mirage* and the corresponding Mirage image format. The Marvin Image Store enhances the design of Mirage with several novel ideas: support for layers and thus special files used by union file systems, Merge and Diff operations solely based on metadata, Direct Mounts as well as exchangeable storage back ends that deploy various compression algorithms to further increase storage efficiency. Those extensions increase both storage and access efficiency and are the foundation of novel management, maintenance, and analysis operations.

The remainder of this section is structured as follows. In Section 5.3.1, a list of requirements for storage of virtual machine images is defined. Section 5.3.2 describes two different approaches to virtual machine storage and introduces the concept of deduplication. In Section 5.3.3, the Mirage system is described and assessed regarding the requirements. Based on this description and the requirements, Section 5.3.4 illustrates the design of the Image Store.

5.3.1 Requirements

The Marvin Image Store needs to satisfy all requirements listed in this section to provide the desired functionality. They are divided in two parts: fundamental requirements that are absolutely necessary to build a storage system for virtual machines and additional requirements that facilitate novel operations on stored virtual machines.

5.3.1.1 Fundamental Requirements

The following list contains the fundamental requirements on a storage system for virtual machines:

R5.1 Provide a storage system that can store a large number of virtual machine images both efficiently and safely.

R5.2 Provide an efficient version control system for stored virtual machines.

R5.3 Provide means to efficiently update a virtual machine stored in the Image Store.

R5.4 Provide fast access to the image of a stored virtual machine (in any version).

R5.5 Provide an efficient way to clone virtual machines from an existing shared image.

R5.6 Provide an efficient way to make minor modifications to a virtual machine image to facilitate deployment of multiple, almost identical instances of the same virtual machine image.

The Image Store has to be able to handle the large number of virtual machine images used in Virtualized Grid and Cloud Computing environments without using excessive amounts of storage space. Without this storage efficiency, the Image Store cannot be used as a virtual machine image repository in such environments. Furthermore, it ensures that the images are stored in a way so they cannot be damaged during normal use of the Image Store (Requirement R5.1).

Virtual machines have to be maintained just like physical machines. Regular maintenance is important even in phases without active usage of a virtual machine (see Section 3.3.2). Consequently, virtual machine images change over time. Just overwriting a stored image with an updated one violates Requirement R5.1, because in case of an error both versions of the virtual machine might be lost. A version control feature (Requirement R5.2) solves this problem by generating a new version of a virtual machine when it is updated instead and preserves the old version. Furthermore, it stores all these versions in a storage efficient way. Finally, it provides tools to manage the different versions of a virtual machine, e.g., a tool that analyzes the changes between two versions.

Besides actually storing multiple versions efficiently, the process of updating a stored virtual machine – the creation of a new version – has to be efficient as well. This is ensured by Requirement R5.3. This requirement is even more general: every update to a virtual machine image should be efficient. This also includes updating a copy of a virtual machine image outside of the Image Store to the latest version available in the Image Store.

The deployment time of a virtual machine is a critical factor for the elasticity and thus the perceived performance of a Cloud Computing environment. A Cloud Computing environment can only provide the desired elasticity if all components in the infrastructure including the virtual machine image repository are optimized for maximum efficiency. Requirement R5.4 ensures that the Image Store provides the necessary access efficiency to minimize the deployment time of virtual machines. The use case of deploying older versions of a virtual machine images is explicitly included in this requirement, because excessive deployment times for older versions limit the usability of the version control feature.

Sharing of virtual machine images between users is an important feature, because it eases the process of creating new virtual machines (see Section 5.3). Consequently, the Marvin Image Store must supports sharing of virtual machine images as well as cloning from shared images in an efficient manner (Requirement R5.5).

Finally, if multiple instances of a single virtual machine have to be deployed, it might be necessary to make minor modifications to the configuration of the virtual machine. An example of such a modification is the change of the hostname that is stored in the file `/etc/hostname`. Although virtual machines should be configured in a way that does not rely on configuration changes when multiple instances of a virtual machine need to be deployed, it is not always possible to do this. The Image Store has to provide an efficient process for this kind of minor changes (Requirement R5.6). Note that this requirement is different from Requirement R5.3 that deals with arbitrary and substantial changes to a virtual machine, e.g., by updating software or installing additional software.

5.3.1.2 Additional Requirements

All the requirements defined above ensure that the Image Store can provide the basic functionality to serve as virtual machine image repository. In the following, a set of additional requirements that support advanced operations on stored virtual machine images is listed.

R5.7 Provide support for storing layered virtual machine images.

R5.8 Provide means for analyzing the contents of stored virtual machine images efficiently.

R5.9 Provide fast access to the contents of any file contained in (any version of) a stored virtual machine.

R5.10 Provide means for filtering of sensitive or unwanted files out of virtual machine images.

The Image Store can be combined with the image composition technology presented in Chapter 4 to achieve better access efficiency by reducing the sizes of virtual machine images. Thus, it has to provide support for storing layered virtual machine images and has to be able to deal with their characteristics (Requirement R5.7).

Being aware of the content of virtual machine images instead of viewing them as plain files enables novel ways to handle virtual machines. The Image Store has to provide means for in-depth analysis of stored virtual machine images (Requirement R5.8). One of the use cases for this kind of functionality is the detection of the differences between two versions of a virtual machine images. This can be used to locate relevant changes in case a software update or another modifications of a virtual machine image causes an error. It can also be used to derive necessary steps to update a cloned image based on the difference between the original image and its latest version. Another use case for this kind of functionality is computing a list of virtual machine images containing either a vulnerable binary or a configuration file that contains insecure settings. After relevant virtual machines have been found, it is important to be able to access the content of the corresponding files. The Image Store has to provide fast access not only to the images themselves, but also to their contents (Requirement R5.9).

Finally, sharing of virtual machine images may lead to the disclosure of sensitive information, if another user clones an image containing sensitive files. Sensitive files include but are not limited to Secure Shell (SSH) [70] related files (i.e., keys, known hosts, configuration files with host specific settings), Transport Layer Security (TLS) [71] key pairs, or shell history files. The creator of the virtual machine image can easily delete these files, but there is always the risk of forgetting this important manual step. Thus, automation of this step is necessary, especially when a shared virtual machine image is regularly updated. Sensitive data, however, is not the only type of data that can and should be filtered out. Temporary files, caches and log files do not need to be stored in a virtual machine image repository. The Image Store must be able to filter both sensitive and unwanted files out from a virtual machine image either when it is stored or updated or before it is shared (Requirement R5.10).

5.3.2 Storage of Virtual Machine Images

This section starts with a short introduction into the concept of deduplication, because it is a very important technology that can be used to reduce the storage requirements of virtual machine images. Then it compares two contrary approaches for storing virtual machines: the *content agnostic* and the *content aware* approach. These approaches deal with virtual machines as black or white boxes, respectively. Both approaches are evaluated regarding the requirements of virtual machine image storage and the results of the assessment are discussed.

5.3.2.1 Deduplication Technology

The goal of deduplication is to increase storage efficiency when storing large amounts of data. It is accomplished by splitting the data into small chunks and identifying duplicate chunks of data. Only a single instance of each unique data chunk is stored, i.e., duplicate chunks of data are stored only once. This can be achieved by using a content-addressable storage system that computes location of a chunk of data from the data itself by applying a hash function to the data. Because identical chunks of data have the same hash value and thus the same address in the content-addressable storage system, they are effectively stored only once. To be able to reconstruct the original data from the chunks, a list of references to the data chunks is stored as well. This obviously increases the storage requirements for unique data chunks, because both the chunk and the reference need to be stored, but decreases the storage requirements as soon as a data chunk appears more than once.

The actual amount of storage space that can be saved using this technique depends on the type of data, i.e., the amount of duplicate data chunks it contains, and the *deduplication granularity*, i.e., the size of the chunks. Deduplication techniques exist for both fixed and variable sized chunks of data. This facilitates the use deduplication with granularities ranging from fine, e.g., small, fixed size blocks of 512 bytes, 4,096 bytes, or 8,192 bytes, to coarse, e.g., entire files. In the latter case, each file's content is stored as a single, variable size chunk of data. The selection of a granularity is a trade-off between deduplication and resource requirements. Smaller chunks of data reduce the storage requirements for the actual data, because the amount of unique chunks of data is reduced. On the other hand, smaller chunks cause an increase in the number of references required to reconstruct the original data. The opposite is true for larger data chunks. Most deduplication systems use chunks of fine granularity, because this saves storage space even for files that are only partially equal.

The image composition approach presented in Chapter 4 also constitutes a form of deduplication. Because every layer is stored as a single chunk of data, it is an example of a very coarse-grained deduplication system. Nevertheless, the image composition approach achieved high savings. The achievable savings of more fine-grained deduplication methods should be even higher.

5.3.2.2 Content Agnostic Storage - A Black Box Approach

The first approach is called content agnostic, because it treats virtual machine images as plain files or black boxes. Different techniques can be used to reduce the storage requirements of virtual machine images in this approach: besides the obvious option of deduplication, compression or differential storage can also be used for storing virtual machine images efficiently. All three techniques do not naively store image files, but rely on similarities within the images that can be utilized to reduce the storage requirements. Regardless of the chosen technique this is a black box approach, because all three techniques treat virtual machine images as simple streams of bytes instead of analyzing their contents and utilizing similarities at the file system level.

Deduplication is a widely used approach to reduce the storage requirements of any kind of data, not only virtual machine images. Various solutions for deduplication exist that can be used to store virtual machine images in a storage efficient way. First of all, some commodity file systems like ZFS [113] or Btrfs [23] natively support deduplication. Furthermore, there are highly specialized deduplication solutions like lessFS [84] or SDFS [145]. All of these solutions provide a file system to their user. The virtual machine images to store are just copied into those file system like regular files and the deduplication is taken care of automatically.

Another very common approach to reduce the storage requirements of data is to compress it using one of the many available compression algorithms. This approach can easily be used for storing virtual machine images. If an image has to be stored, it is compressed using the selected algorithm like any other file. Before a virtual machine can be started, the corresponding image has to be decompressed again.

Differential storage technology is often used in backup programs and can also be used to store virtual machine images. The idea of differential storage for virtual machine images is to store the first version of the image in its entirety and instead of the entire image of the second version only to store the parts of the image that have changed between the two versions. This can be carried on, e.g., instead of the third version's image only the changes between the second and third versions' images are stored. Obviously, this technique can only store multiple versions of a single virtual machine's image efficiently. The bigger the differences between to images, the smaller the storage savings. Fortunately, cloned images are nothing else than another version of a virtual machine image, although there is no longer a linear version history, but a tree-like version history.

The drawback of this approach is that in many cases images have to be reconstructed before they can be used. If the third version of the image shall be used in this specific case, the first version has to be copied and the changes between the first and second image and between the second and third image have to be applied to the copy in this order. There are many variations of this technology that improve the access time to specific versions, e.g., storing complete images every n versions or storing the last image in its entirety and store the differences to older versions. Unfortunately, not all of these variations are suitable for tree-like version histories.

An assessment of the content agnostic approach for storing virtual machine images is given in Table 5.1. It at best partially satisfies the fundamental requirements imposed on a storage system for virtual machines. It fails to satisfy the additional requirements that support advanced operations on stored virtual machine images.

5.3.2.3 Content Aware Storage - A White Box Approach

The second approach is tailored to storing virtual machine image files (or file system images in general). It does not treat virtual machine images as plain files, but analyzes them at the file system level and uses the gained knowledge of the contents to reduce the storage requirements. In other words, it deals with the individual files in the virtual

Table 5.1 Assessment of Content Agnostic Storage. An assessment of a black box approach to virtual machine image storage.

Requirement		Assessment of Content Agnostic Storage
Fundamental Requirements		
R5.1	✓	Large numbers of virtual machine images can be stored efficiently using either deduplication, compression or differential storage. The safety of the system is subject solely to the actual implementation.
R5.2	(✓)	A black box approach can provide safe storage for multiple versions of a virtual machine (Requirement R5.1). However, none of the advanced features of a version control system are available.
R5.3	×/(✓)	A black box approach obviously has to update images in their entirety. The efficiency such an update additionally varies depending of the implementation technology.
R5.4	(✓)/✓	Efficient access to virtual machine images in any version is possible depending on the implementation technology. Of the examples above, only the deduplication and compression technology provide efficient access.
R5.5	×/(✓)	The entire disk image has to be copied to create a clone. Depending on the actual implementation the copy process might be more efficient than a regular copy process.
R5.6	×	Even minor modifications cannot be done in a black box approach, because the storage system knows nothing about the content of the image.
Additional Requirements		
R5.7	✓	Layered virtual machine images can be stored just the same as regular virtual machine images.
R5.8	×	A black box approach does not provide any means for analysis of virtual machine's contents.
R5.9	×	A black box approach does not provide any means for directly accessing virtual machine's contents without accessing the entire image.
R5.10	×	A black box approach does not provide any means for filtering of virtual machine's contents.

✓: requirement satisfied, ×: requirement not satisfied, (✓) : requirement partly satisfied

machine image and not the image file as a whole. This approach is consequently called content aware.

The content aware approach is based on the idea of discarding the image file and the file system it contains, because it is just a container and has not to be stored. Only the files contained in the image are stored by this approach. If the file system is discarded, the structure of the files, i.e., the hierarchy of directories, is lost. This structure has to be captured in special file called *manifest* and stored with the files. Together, manifest and files form a blueprint to reconstruct the image file on demand.

Up to now, the storage requirements are not significantly reduced[1]. This goal is achieved using a content-addressable storage system that is the natural continuation of this approach, because the individual files need to be stored in an appropriate manner anyway. The manifest contains a reference for each file that can be used to retrieve it.

To be able to analyze the contents of a virtual machine image, fast access to the metadata of all files contained in the image is required. The metadata comprises among others the name of a file, access and modification timestamps, access rights and owner information. On the other hand, the data is the actual content of the file. If the metadata of all files is stored in the manifest together with the structure, the manifest contains the entire metadata of the virtual machine image. That way, the content-addressable storage system stores only the files' contents, but none of their metadata.

At this point, the content aware approach combines a clear separation of data and metadata as well as a deduplication of file contents. The granularity of the deduplication can be chosen freely in the implementation of the approach. Because individual files are stored instead of large virtual machine images, the content-addressable storage may even use a coarse granularity to reduce the overhead, i.e., an entire file is a chunk.

An assessment of the white box or content aware approach for storing virtual machine images is given in Table 5.2. It can completely satisfy both the fundamental as well as the additional requirements imposed on a storage system for virtual machines. However, note that this is a theoretical evaluation of an approach. An implementation of this approach can nevertheless fail to satisfy all requirements.

5.3.2.4 Discussion

A comparison of the suitability of both presented approaches for the composition of disk images is given in Table 5.3. It is obvious that the content agnostic approach is not suited very well for storing virtual machine images when the requirements are not limited to plain storage of images, but cover advanced features like analysis of images or version control. The content aware approach can in theory satisfy all of the requirements imposed on a storage system for virtual machines. It is thus the obvious choice for the Image Store.

5.3.3 Basic Concepts

In this section, the basic concepts the Image Store is built upon are presented. These concepts are based on the work of Reimer et al. [128]. The novel ideas that differentiate the Image Store from these basic concepts are presented in Section 5.3.4. Throughout the description of the concepts, satisfied requirements are indicated in parentheses.

[1] Except the image file is much larger than the contents of the image file, which is typically the case for fixed-size or non-sparse images.

Table 5.2 Assessment of Content Aware Storage. An assessment of a white box approach to virtual machine image storage.

Requirement		Assessment of Content Aware Storage
Fundamental Requirements		
R5.1	✓	Large numbers of virtual machines images can be stored efficiently because of the content-addressed storage system for file contents that guarantees their safety at the same time. The safety of the manifests is subject to the implementation.
R5.2	✓	A white box approach can provide safe storage for multiple versions of a virtual machine (Requirement R5.1). Additionally, the content awareness is a prerequisite for providing advanced version control system features.
R5.3	✓	A white box approach can be used to facilitate efficient updates, but this depends solely on the implementation.
R5.4	✓	Efficient access to virtual machine images depends on efficient access to both the manifest and the file contents. Both can be achieved by an implementation.
R5.5	✓	Only the manifest needs to be copied to clone a virtual machine. This is very efficient because manifests are small compared to virtual machine images.
R5.6	✓	Minor modifications to virtual machines can be done efficiently, because the white box approach has knowledge about the images' contents.
Additional Requirements		
R5.7	✓	The content aware approach can be used to support layered virtual machines, but this depends solely on the implementation.
R5.8	✓	The separation of data and metadata used in the white box approach is the ideal foundation for analyzing virtual machine images, because it facilitates working solely on metadata that is smaller an can be processed more efficiently.
R5.9	✓	Efficient access to individual files' content is trivial with this approach: the user looks up a reference in the manifest and fetches the contents from the content-addressable storage.
R5.10	✓	Content awareness is a precondition for filtering. However, the filtering capability depends solely on the implementation.

✓: requirement satisfied

5.3.3.1 Import and Export of Virtual Machines

The Image Store is an implementation of the content aware approach that works with the contents of image files instead of the image files themselves (see Section 5.3.2.3). Thus, a virtual machine image file cannot be just copied into the Image Store, but it has to be imported. During the import process (depicted in Figure 5.2), the files are extracted from the image file while the image itself is discarded. Furthermore, the Image Store separates the actual data from the metadata, i.e., it separates the content

Table 5.3 Suitability of Storage Approaches. Comparison of the numbers of satisfied, partly satisfied and non-satisfied requirements for each of the two approaches to virtual machine storage.

	Satisfied		
Approach	Fully	Partly	Not at All
Content Agnostic	2 (3)	2 (3)	6 (4)
Content Aware	10	0	0

of the files in the image from file names, access rights, timestamps, owner information, and the file hierarchy. The data is kept in the *Data Store*, a content-addressable storage system, while the metadata of each image is written into a *manifest* stored in the *Metadata Store*. The connection between Metadata Stored in the manifest and the corresponding chunks of data in the Data Store is realized via references based on the chunk's content. A single chunk of data can be referenced by either multiple manifests or multiple times from within a single manifest. Thus, the Data Store is a deduplication system and enables efficient storage of virtual machine images (Requirements R5.1 and R5.2).

The Image Store treats entire files as chunks of data and thus applies deduplication with a coarse granularity, although it could also be implemented with finer grained deduplication. When considering the typical lifecycle of virtual machines in Virtualized Grid and Cloud Computing environments and its implications (see Section 3.3.2) the decision in favor of this coarse granularity is nevertheless reasonable. Virtual machines contain mostly binaries and configuration files, but no data. On the one hand, data in a virtual machine would vastly increase its deployment time. On the other hand, when a virtual machine that contains older data is deployed inconsistencies with other instances can occur. More importantly, in Virtualized Grid and Cloud Computing environments the disks associated with running virtual machines are *ephemeral*, i.e., their contents are lost when a virtual machine is shut down. Therefore, data should not be stored on the system disks of virtual machines[2], but on external object or block storage systems, e.g., the Simple Storage Service (S3) and Swift or Elastic Block Store (EBS) and Cinder in case of Amazon Web Service [8] and OpenStack [111], respectively. As the binaries are very likely identical for many virtual machines – after all there are only a few different Linux distributions that are used in most of virtual machines – the coarse granularity is beneficial because less references and thus lookups in the content-addressable storage are required to reconstruct the files during export.

A manifest is a document that contains the metadata for each file in a virtual machine image and references to the content of those files. The metadata of a file is comprised of the file's name, size, access rights, ownership information and a few additional attributes. Except for the file name the metadata is taken directly from the *index node* (inode), i.e., the data structure that represents a file system object in Unix-style file systems. The manifest itself is a tree structure that corresponds to the hierarchical

[2] The system disk of a virtual machine is a copy of the image file when the virtual machine is started, but diverts from the image file during the usage of the virtual machine.

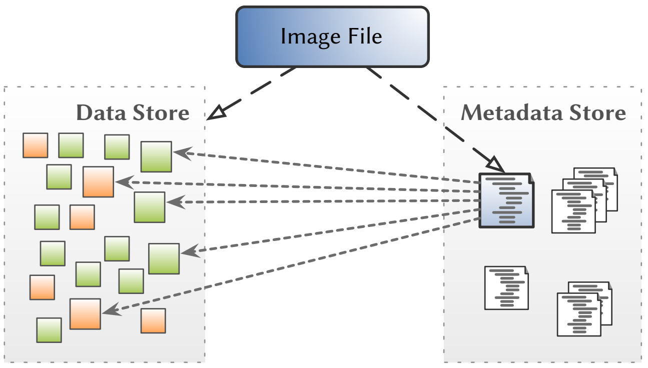


Figure 5.2 Import of a Virtual Machine Image. During the import process, the image file is split into its metadata, consolidated in a manifest in the Metadata Store, and its files, stored in the Data Store and referenced by the manifest. The Data Store contains both files shared between multiple virtual machines as well as virtual machine specific files, illustrated using green and orange boxes, respectively.

tree structure of the file system it represents. The metadata for each of the files is stored in a tree node together with a reference to the file's contents stored in the Data Store.

The manifests of all virtual machine images stored in the Image Store are kept in the Metadata Store. For each of the virtual machines, a single or multiple manifests are stored. Each manifest is the blueprint for a specific version of a virtual machine. The Metadata Store ensures that stored manifests are immutable. When a virtual machine is updated and subsequently imported again, the original manifest is not modified, but a new manifest is created for the updated virtual machine. Because the metadata accounts only for a small part of size of a virtual machine image, that approach is acceptable. This is one of two preconditions necessary to provide version control for virtual machines (Requirement R5.2). For better access efficiency, the manifests are stored in a self-contained manner. They are therefore immediately available and do not have to be computed by applying a (potentially large) number of differential manifest updates to an initial version of the manifest (Requirement R5.4).

The contents of all files that are contained in any of the stored virtual machine images are kept in the Data Store. For each unique file, the entire content of the file is stored as a single object. The Data Store is an instance of a content-addressable storage, i.e., the content of the file is used to calculate the position in the Data Store (Requirement R5.9). The position is stored in the manifest to reference the content. Using the content-addressable storage approach serves two purposes: it provides the deduplication of file contents and guarantees the immutability of file contents stored in the Data Store. The immutability of file contents is the second precondition necessary to provide version control for virtual machines (Requirement R5.2). It is guaranteed by an inherent property of a content-addressable storage: as soon as a file is modified, i.e., its contents change, another position is calculated so the updated file's content is stored somewhere else in the Data Storage leaving the original content untouched.

Obviously, a virtual machine image stored in the Image Store cannot be used by a virtual machine directly, because it is not stored in the form of an image file. It has to be exported before usage, i.e., reconstructed from the information in the manifest stored in the Metadata Store and the referenced file contents in the Data Store. The export of an image consists of two steps. First, a new image file containing an empty file system is created. Then, for each node in the manifest a new file is created in the image, the referenced contents are copied to the image from the Data Store and finally the metadata from the manifest is applied, i.e., the access rights, owner information and timestamps are set to the values stored in the manifest. The exported image is almost identical to the one that was imported into the Image Store[3]. The export process is illustrated in Figure 5.3.

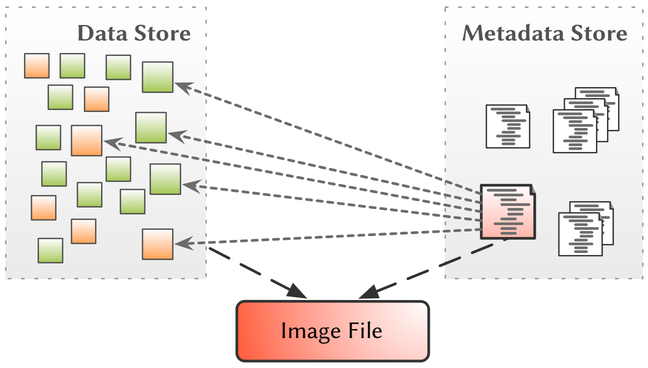


Figure 5.3 Export of a Virtual Machine Image. During the export process, the image file is reconstructed using the manifest in the Metadata Store. The referenced files are copied from the Data Store to the target file system and the metadata contained in the manifest is applied.

5.3.3.2 Sharing and Cloning of Virtual Machines

Sharing of virtual machine images is an important concept in both Virtualized Grid and Cloud Computing. It allows users without detailed knowledge of (virtual machine) administration to deploy their own virtual machines in the Cloud or to execute jobs in Virtualized Grid Computing environments by relying on existing virtual machine images created by experts. To facilitate modifications of a shared image, it needs to be cloned first. Depending on the size of the virtual machine image cloning is a time-consuming process with traditional image files. On the other hand, using the Image Store cloning is a fast process: only the manifest needs to be copied to generate a clone of the virtual machine. This is significantly faster, because the manifest's size is only a fraction of image file's size (Requirement R5.5).

Another related use case is the process of deploying multiple instances of a virtual machine, if some of its files need to be slightly modified before deployment to adopt the virtual machine's configuration. Without any optimizations, this would require to clone the image, export the cloned version, make the necessary changes, and finally updating the clone in Image Store. To speed up this process, a technique named

[3] Details about the (subtle) differences are given in Section 5.4.3.2.

manifest composition [128] can be used. The idea is to create an *overlay manifest* containing only the modified files. The overlay manifest is then combined with the manifest of the virtual machine to create a modified version. This approach is significantly faster compared to the non-optimized approach (Requirement R5.6).

The final concept closely linked with sharing of virtual machines is filtering of sensitive information as proposed by Wang et al. [167]. One of the problems related with sharing of virtual machine images is that they may contain sensitive content, such as SSH keys, SSL key pairs, or shell history files. Using a set of filters that are executed either at the time a virtual machine image is shared or exported, sensitive content is removed (Requirements R5.8 and R5.10). The owner of the virtual machine image repository provides the filters. Users can specify additional filters using high-level transformation rules based on regular expressions that are applied to the content of the files. No user-supplied code is executed to implement a filter for security reasons. Because of this restriction, custom filters are less powerful than the provided ones, although they are important because the creators of shared images typically have more knowledge about the sensitive content that should be removed.

5.3.3.3 Discussion

An assessment of a basic Image Store based on the concepts described above, which is based on the work of Reimer et al. [128] and Wang et al. [167], is shown in Table 5.4. It can be seen that there are both a fundamental and an additional requirement that are not satisfied by this basic implementation: efficient updates and support for layered virtual machines. Updates of virtual machines are possible with the Image Store described above, but the process of updating is not very efficient. Even if only a single file has been changed, the whole image needs to be imported again. Duplicate files will be recognized during the import process and only the modified file will be added to the Data Store, but nevertheless every file must be checked during the process. Layered virtual machine images contain some special files that are necessary to delete files or directories from lower, read-only layers. While these special files can be exported just a regular files – notably they are regular files with a special meaning for the union mount implementation only – the basic Image Store is not aware of their function. This has severe consequences for *manifest composition* and advanced operations.

Furthermore, there are four requirements that are only partially satisfied. Different versions of virtual machines are stored, but there is no way to work with these different versions. Access to virtual machine images stored in the Image Store requires the entire image to be exported, unless a single file is fetched individually from the Data Store. By reducing the amount of data that needs to be exported, the access time can be further improved. The analysis capabilities of such a store are limited to finding specific content in virtual machines. Finally, the filtering solution provides only a rudimentary extension mechanism for security reasons.

From the assessment it can be seen that a basic Image Store is not the solution for all challenges related to storing virtual machine images. In the next section, a revised version of this concept is proposed. It incorporates novel ideas to satisfy the remaining requirements.

Table 5.4 Assessment of a Basic Image Store. An assessment of the suitability of a basic storage system based on work of Reimer et al. [128] and Wang et al. [167] as a virtual machine image repository.

Requirement		Assessment of a Basic Image Store
Fundamental Requirements		
R5.1	✓	Deduplication as well as immutable data and manifests guarantee that images are stored both efficiently and safely.
R5.2	(✓)	The storage concept automatically preserves older versions (see Requirement R5.1), so the basic functionality of a version control system is provided. None of the more advanced features of commodity version control systems are provided.
R5.3	×	Updating a virtual machine requires to export and reimport the entire virtual machine irrespective of the amount of changes.
R5.4	(✓)	Self-contained manifests for different versions enable fast access to every version of a stored virtual machine, but images can only be exported in their entirety except for fetching individual files (see Requirement R5.9).
R5.5	✓	Cloning is very efficient, because only the manifest needs to be copied without ever touching actual data.
R5.6	✓	Overlay manifests enable very efficient minor modifications of virtual machines compared to full-blown updates (see Requirement R5.3).
Additional Requirements		
R5.7	×	Layers can be imported into a basic Image Store, but it is not aware of and cannot handle the specifics of layers.
R5.8	(✓)	The separation of data and metadata and the use of manifests lays the foundation for a more detailed analysis of the content of images. Except for searching virtual machines for specific content, no further functionality is available.
R5.9	✓	Individual files can be accessed without mounting the image just by retrieving the references from the manifest (in the desired version) and fetching the corresponding files from the Data Store.
R5.10	(✓)	The filter solution is able to remove sensitive files from virtual machine images. Only rudimentary filter extensions are possible for the user.

✓: requirement satisfied, ×: requirement not satisfied, (✓) : requirement partly satisfied

5.3.4 Proposed Solution

In this section, the design of the Marvin Image Store is presented. It is based on the description of the basic concepts in Section 5.3.3 and includes novel ideas that help satisfying the all requirements listed in Section 5.3.1. First, four novel ideas are presented: support for layered virtual machine images at the level of semantic understanding of composition-related artifacts in layers, Merge and Diff operations that enable real version control for virtual machine images, mounting of virtual machine images stored in the Image Store, and exchangeable storage back ends for improved storage efficiency. Afterwards, applications of these ideas that solve concrete problems identified in the last section are described: efficient updates of virtual machines and an extensible filtering mechanism. Finally, an overview of the architecture of the Image Store is given the proposed solution is assessed with regard to the requirements.

5.3.4.1 Layered Image Support

The idea behind the virtual machine image composition approach proposed in Chapter 4 was to use a common base layer that contains the parts of a virtual machine image that are shared by multiple virtual machine. The remaining parts are part of one or more additional layers, i.e., a user layer, a vendor layer, or both, which are combined with the base layer by the Image Compositor. The size of the additional layers containing the virtual machine's specific parts is significantly smaller compared to a traditional virtual machine image containing both shared and specific parts. Furthermore, base layers are always used in read-only mode[4] and can thus be cached on the execution host. In this case, only the additional layers needs to be transferred to the execution host during the deployment of a virtual machine.

When the virtual machine to deploy is stored in the Image Store, the export process is part of the deployment phase. In this case, the combination of virtual machine image composition and Image Store (Requirement R5.7) is useful as the export time of the virtual machine is reduced if only the additional layers need to be exported instead of the entire virtual machine image. Therefore, combining these technologies can increase the access efficiency of the Image Store (Requirement R5.4).

The Image Compositor uses the union mount technology to compose the root file system of a virtual machine using multiple layers. Two fundamental rules are applied by union mounts: files in a higher layer replace files or directories in lower layers and directories in a higher layer merge their contents with their lower layer counterpart's contents. Special markers, so-called *whiteout files* [173, 10, 11, 108] are created in the higher layer to mark files or directories in lower layers as "deleted". Another type of marker is used to mark directories as *opaque* [10, 11] and prevent them from merging of their contents with their lower layer counterpart's contents. This facilitates replacing a directory in a lower layer with an entirely new directory. Write access to the union mount is possible using this rules even if the lower layers are not writable.

A special treatment of whiteout files and opaque directory markers is not required in order to store and reconstruct layer image files correctly, because they are regular files with a special meaning for the union mount implementation only. For advanced features like the Merge and Diff operations (see Section 5.3.4.2) and the analysis of virtual machine images' contents, however, special treatment of both whiteout files and opaque directory markers is required. They are consequently identified during the import process and stored appropriately in the manifest so they can be handled according to their function. Furthermore, by making the Image Store aware of those

[4] Except for situations when the base layer itself should be modified, e.g., to install software updates.

special files and their meaning it is possible to generate the right files for any available union mount implementation during export. This even enables an easy migration of layered virtual machines in case the Image Compositor replaces the union mount implementation with another one.

Besides reducing the size of images to be exported, the support for layered virtual machines is also used to implement efficient updates and advanced content filtering techniques. Details are given in Sections 5.3.4.5 and 5.3.4.6.

5.3.4.2 Merge and Diff Operations

In the description of basic concepts (Section 5.3.3) the manifest composition technique has been briefly mentioned. It was introduced by Reimer et al. to combine overlay manifests with the manifest of a virtual machine in order to quickly generate a clone of a virtual machine with a few modified files. More specifically, it generates a new version of the manifest that contains the modified versions of the files instead of the original ones. It facilitates replacing the content of files, but not to delete files from the manifest. The manifest composition technique therefore is a simplified version of the image composition or union mounts, respectively.

The *Merge* operation is a considerably enhanced version of manifest composition. Like the latter, it combines two manifests to create a new manifest. It borrows the concept of whiteout files and opaque directories from union mounts (see Section 5.3.4.1) to support the deletion of files and directories from the resulting manifest. The semantics of the Merge operation are identical to the semantics of union mounts, so both approaches can be used together. The contents of a union mount of a base layer L_B and a user layer L_U are thus identical to the result of a merge operation of the corresponding manifests M_B and M_U. Obviously, the Merge operation cannot be commutative, because files in M_U replace files in M_B, like files in L_U replace files in L_B when a union mount is created.

The *Diff* operation can be used to determine the differences between two manifests M_1 and M_2. This operation is especially useful if M_1 and M_2 are different versions of the same virtual machine, because it can be used to analyze the changes between different versions (Requirement R5.8). Notably, the result of the Diff operation is not a plain list of changes, but a new manifest. For every file that is changed between M_1 and M_2, the resulting manifest contains the changed metadata and a reference to the modified content from M_2. If a file does only exist in one of the two manifests, it will be included in the resulting manifest or replaced by the manifest representation of a whiteout file depending on whether it is existing in M_2 or M_1, respectively. Consequently, like the Merge operation, the Diff operation cannot be commutative.

Basically, the resulting manifest M_R is a blueprint of how to modify M_1 to become like M_2. If M_1 is merged with M_R using the Merge operation described above, the resulting manifest $M_{R'}$ will be equal to M_2.

The Merge and Diff operations can not only be used for analyzing virtual machine images' contents, but are also used to implement efficient updates and advanced content filtering techniques. Details are given in Sections 5.3.4.5 and 5.3.4.6.

5.3.4.3 Direct Mounts

Manifests facilitate fast and easy access to the metadata of virtual machines, e.g., to determine which virtual machines contain a specific file either by its path or content. A potential use case is to determine quickly all virtual machines that a specific security update has been installed in. In this case, the virtual machines' manifests are searched for a specific path or a file that references a specific chunk of data in the Data Store, instead of trying to interpret the package databases contained in the images. This kind of search can be used for single virtual machines and easily be extended to a larger set of virtual machines. The content of files found using one of the methods just described can then be fetched from the Data Store.

While this approach to analyzing virtual machines' contents clearly has advantages when the number of virtual machines to analyze is high, there is a severe drawback. Metadata searches and fetching of individual files is not the natural way to interact with the contents of a virtual machine. Typically, a user would mount a virtual machine image and use arbitrary tools to access its contents. With the Image Store described in Section 5.3.3 this would require that the image is exported first. The Marvin Image Store provides a *Direct Mount* feature. It is able to mount a stored virtual machine image in read-only mode without exporting the image first. A user can then use standard tools to find the relevant information inside the virtual machine image. Furthermore, this approach can be used to inspect a single virtual machine and find the paths or references[5] for searching a larger set of virtual machines. Additional use cases for Direct Mounts are described in Section 5.4.6.

5.3.4.4 Exchangeable Data Store Back Ends

The Data Store is responsible for safely keeping the content of files. It does so by implementing a content-addressable storage that both provides deduplication to increase the storage efficiency and guarantees the immutability of the file contents. To further improve the storage efficiency beyond what the deduplication provides, the Data Store supports exchangeable back ends that leverage different techniques to reduce its storage requirements (Requirements R5.1 and R5.2). By using a modular design, the creation of new back ends is easy, so new techniques can be easily integrated. Besides a back end that stores the content unchanged, there are a *sparse* back end and a set of *compression* back ends.

The sparse back end tries to detect large blocks of zeros in the data to store. If the underlying file system supports sparse files, it stores the data in sparse files skipping the blocks of zeros.

The compression back ends utilize different compression algorithms to compress the data before it is stored. This approach is superior compared to compressing entire virtual machine images, because the metadata is immediately available in the Metadata Store. Therefore, search or analysis operations can be executed without

[5] As described in the implementation section, the references are based on the hash value of the chunk of data they address. A user can easily calculate references manually (see Section 5.4.2.2).

decompressing the data. Only if the data needs to be accessed, e.g., during the export of a virtual machine, decompression is necessary and a slightly reduced access efficiency is expected. Compression is only required on the first time a chunk of data is stored, so the import process is usually not affected by compression.

5.3.4.5 Efficient Virtual Machine Updates

Undoubtedly, continuous maintenance of virtual machines is very important (see Section 3.3.2.1). Whenever software updates have to be installed in a virtual machine stored in the Image Store described in Section 5.3.3 the three steps depicted in Figure 5.4 have to be executed. In Step 1, the image file of the virtual machine to be updated is exported from the Image Store to an arbitrary host. Afterwards, the virtual machine is started, the software updates are installed in the virtual machine as usual, and the virtual machine is shut down in Step 2. Finally, the updated image file of the virtual machine is imported back into the Image Store in Step 3. The resulting manifest describes the new version of the virtual machine that likely shares many files with the previous version. These three steps are required whenever the virtual machine is changed either for installing new software, updating software that is already installed, or just changing the configuration of the virtual machine.

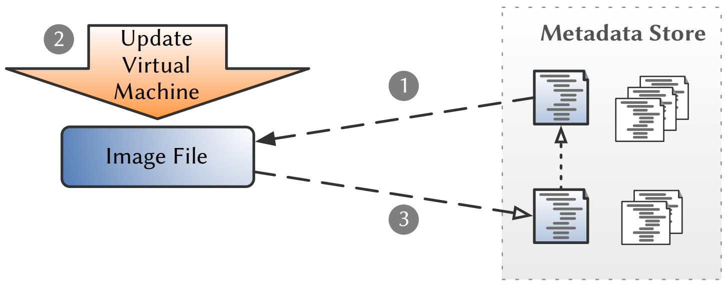


Figure 5.4 Basic Virtual Machine Image Update Process. The non-optimized process of updating a virtual machine image stored in the Marvin Image Store consists of three steps. In Step 1 the image file is exported. Afterwards, the virtual machine is updated in Step 2. Finally, the updated image file is reimported in Step 3. The dotted arrow depicts the version control feature: it points from the current to the preceding version of the manifest. (Data Store omitted.)

The import process (Step 3) is identical to the initial import process. Again, the entire image including the content of all files it contains has to be examined by the Image Store, although only the content of modified files is copied to the Data Store. Obviously, this approach for updating virtual machines stored in the Image Store is not very efficient, especially if the changes are small compared to the size of the virtual machine image. If software is updated frequently, this is very likely the case.

Using either image composition – and thus layered virtual machine images – or the Merge and Diff operations, the import process of an updated virtual machine can be accelerated significantly by reducing the amount of data the Image Store has to examine (Requirement R5.3). The former approach uses image composition to redirect all write accesses to a dedicated layer, which afterwards only contains the files that were changed. Only this dedicated layer is then imported. The second approach splits the import task in two phases: a *metadata import phase* and the *data import phase* that only examine the metadata and the data of the image file, respectively. The metadata input phase generates a manifest that does not contain references to the file contents but is otherwise complete. Using the Diff operation, the files that were changed are determined and only those files are imported in the data import phase.

The export process (Step 1) can also be accelerated if the exported image file of an older version of the virtual machine still exists and has not been changed since the export. This can be guaranteed using image composition that prohibits write access to image files. The Diff operation is used to determine the differences between the older and the current version of the virtual machine. Only the files that have been changed since the export of the virtual machine image are exported to a dedicated image file. A virtual machine in the current version can then be created with the Image Compositor using the older image and the image containing the differences.

It is important to note that the optimized export process using the Diff operation is not only useful when a virtual machine stored in the Image Store is updated, as described above. On the contrary, this approach can be used to speed up every export process. It enables efficient virtual machine updates in *both directions*: from the virtual machine image to the image store and the other way around.

5.3.4.6 Advanced Content Filtering

As already stated in Section 5.3.3.2, filtering of the contents of virtual machine images is important when those virtual machines are shared, because it facilitates deleting of sensitive content from those images before it gets in the wrong hands. Unwanted files are another category of files that the Image Store has to handle. There is no harm if a virtual machine containing such files is shared, because they are not sensitive. Instead, unwanted files occupy space in the Data Store and slow down import and export processes. Examples of unwanted files are temporary files and log files that are stored in the virtual machine at the time of import. These files are not required for the virtual machine to function, so they can be safely removed. Additionally, this kind of files very likely has unique content and thus their storage requirements cannot be reduced by the coarse grained deduplication approach implemented in the Image Store. Other examples of unwanted files are some of the files common package management solutions cache: software archives and repository databases. These files are less problematic, because they are shared between a high number of virtual machines. Nevertheless, removing those files should cause no problems, because those files can be downloaded again – in case of the package database, they certainly will be downloaded again.

The Image Store deals with sensitive and unwanted files by providing content filtering techniques for manifests (Requirement R5.10). It provides two different types of filters: *Metadata Filters* and *Live Filters*. Metadata Filters work solely on the metadata contained in the manifests of virtual machines. They facilitate selecting or rejecting files based on arbitrary criteria, i.e., paths, timestamps, or ownership information.

This type of filter is very fast, but obviously also kind of restricted, because it does not work on the contents of the files.

Live Filters, on the other hand, are scripts that work directly on the virtual machine image. Using the image composition technique, the image file is mounted together with a dedicated layer that captures all write requests. In this way, the scripts used to scan the image files for sensitive or unwanted content can utilize arbitrary tools. A dedicated virtual machine can be used to safely execute user supplied Live Filters. Contrary to Metadata Filters, a Live Filter is not only able to delete file, but also to modify files. Both the image file and the dedicated layer are imported and combined with the Merge operation to apply the changes the filter made.

The Image Store can apply content filters on various occasions: during import, before an image is shared or before export. Filtering during import has the advantage of preventing unwanted and sensitive content from ever being stored in the Data Store. On the other hand, it is inconvenient if the owner wants to update his virtual machine, because the sensitive files are useful for him. The pros and cons are reversed for filtering before a virtual machine is shared. Finally, filtering before export is a particularly bad idea, because it increases the deployment time while it does not have any advantages over filtering before an image is shared. The Image Store facilitates fine-grained control of the execution of filters to deliver the desired results. Removing unwanted file during import and sensitive files before an image is shared seems to be a good compromise between security and convenience.

5.3.4.7 Architecture

The architecture of the Marvin Image Store is shown in Figure 5.5. It consists of the actual storage system, the central component of the Marvin Image Store, and a set of tools for using it. As described in Section 5.3.3.1 the storage system is divided into two parts: the Data Store that stores the contents of all files of the imported virtual machine images and the Metadata Store that stores one manifest for each version of a virtual machine image that has been imported.

The Image Store can be accessed locally from the host it is running on (MIS Server) using this Image Store Manager `mismgr`. It provides the full functionality that has been described above, e.g., importing, exporting, updating, sharing, cloning or mounting of virtual machines, as well as operations of manifests, e.g., merging, calculation of differences, or searching for content. Additionally, `mismgr` contains maintenance functionality for the Image Store itself.

For deployment on execution hosts a prototypical remote client `misrcl` and a dedicated network daemon `missrv` have been developed. The functionality of `misrcl` is currently limited to remotely exporting virtual machines directly on the execution host. Without the remote client, virtual machine images have to be exported on the MIS Server and then transferred to the execution host. The client can browse a list of available virtual machines and their versions and retrieve manifests. With a manifest, `misrcl` is able to create the image file of the corresponding virtual machine on its own by retrieving the content of referenced files directly from the Data Store using a

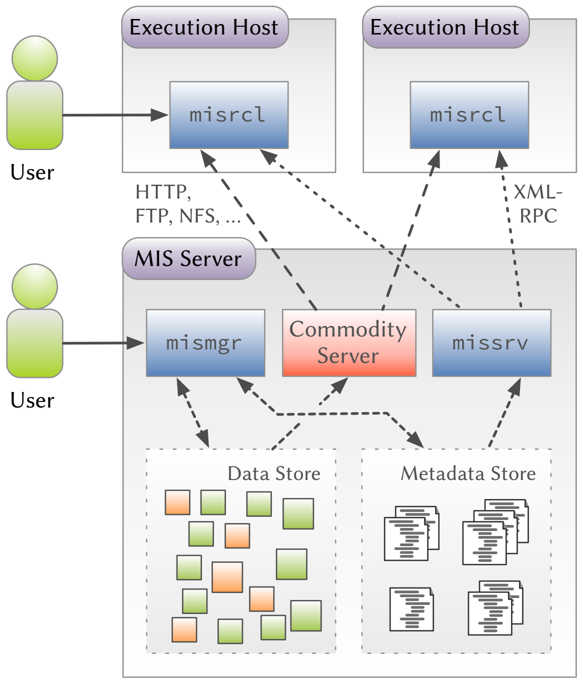


Figure 5.5 Architecture of the Marvin Image Store. The MIS Server with the Data and Metadata Store is the central component of the Marvin Image Store. User can directly interact with the Image Store locally using the `mismgr` tool. Remote access to the Image Store – restricted to exporting images – is provided by the `missrv` and `misrcl` components.

commodity network protocol and appropriate server software on the MIS server. A commodity network protocol is used due to the prototypical nature of `misrcl`. The restriction of its functionality to remote export is a conscious decision, because there is no real access control when accessing the Data Store via such a protocol. With this restriction read-only access to the Data Store is sufficient for the client and thus Requirement R5.1 is not violated even with no advanced access controls. Nevertheless, for real world use either a dedicated protocol for file retrieval from the Data Store that supports fine-grained access control or the implementation of a server push approach is required to prevent leakage of sensitive information.

5.3.4.8 Discussion

An assessment of the Marvin Image Store is shown in Table 5.5. The novel ideas proposed to extend the basic concepts described in Section 5.3.3 ensure that all requirements are satisfied. Especially the combination of the Image Store with image composition and the definition of the Merge and Diff operations improve the access efficiency of the Marvin Image Store by enabling faster update procedures for both stored and exported virtual machine images and enable advanced user-defined content filtering. The Merge and Diff operations and the Metadata Filters additionally enable novel analyses of virtual machine contents. Direct mounts are a useful to support users analyzing the contents of virtual machine images. Finally, the exchangeable storage back ends further improve the storage efficiency.

Table 5.5 Assessment of the Marvin Image Store. An assessment of the suitability of the Marvin Image Store as a virtual machine image repository.

Requirement		Assessment of the Marvin Image Store
Fundamental Requirements		
R5.1	✓	Deduplication as well as immutable data and manifests guarantee that images are stored both efficiently and safely.
R5.2	✓	The storage concept automatically preserves older versions (see Requirement R5.1). Together with the Merge and Diff operations (see Requirement R5.8) this turns the Image Store into a full-fledged version control system for virtual machines.
R5.3	✓	The optimizations of the update process facilitate efficient updating of virtual machine images even for smaller changes.
R5.4	✓	Self-contained manifests for different versions enable fast access to every version of a stored virtual machine. Exports of entire images can be avoided using the optimized export process.
R5.5	✓	Cloning is very efficient, because only the manifest needs to be copied without ever touching actual data.
R5.6	✓	The Merge operation facilitates very efficient minor modifications of virtual machines.
Additional Requirements		
R5.7	✓	The Image Store fully supports layered images. It is aware of the special markers used in layered images and handles them appropriately.
R5.8	✓	Based on the separation of data and metadata and the use of manifests, different analysis operations are available: the Merge and Diff operations, select and reject filters for manifests as well as a search function.
R5.9	✓	Individual files can be accessed without mounting the image just by retrieving the references from the manifest (in the desired version) and fetching the corresponding files from the Data Store.
R5.10	✓	Content filtering is supported using two types of filters. Both types are fully supporting user-defined filter criteria.

✓: requirement satisfied

5.4 Implementation

In the following sections, selected parts of the implementation of the Image Store are presented. It was initially implemented in the Python programming language, Version 2.6, using standard libraries as well as the lxml and llfuse libraries, Python bindings to libxml2 and libfuse2. It is compatible to Version 2.7 of the Python language and the latest versions of the external libraries.

The remainder of this section is structured as follows. In Section 5.4.1 the fundamental structure, in-memory data structure and storage formats of manifests are described. Section 5.4.2 contains a description of the storage system consisting of Metadata Store and Data Store. Afterwards, in Section 5.4.3 the import of virtual machines into and the export of virtual machines out of the Image Store are described. The implementation of three important operations on manifests is shown in Section 5.4.4. Section 5.4.5 contains a description of the Direct Mount feature of the Image Store. The section is concluded with the description of the advanced operations the Image Store provides: efficient update and filtering mechanisms (Section 5.4.6).

5.4.1 Anatomy of Manifests

This section describes the structure of manifests. It starts with a brief introduction into file systems in Linux with a focus on inodes and directory entries that are the blueprint for the manifest structure. Based on this introduction the outline of a basic manifest implementation is presented that is then extended to support layered virtual machine images. Finally, the data structures used store manifests both in memory and on disk as described.

5.4.1.1 File System Overview

In Linux systems, there is a file system abstraction called *Virtual File System* (VFS) [130]. This abstraction constitutes the basic concepts of all Linux file systems that are described in the following[6]. The VFS is also the basis for the definition of manifests.

Index nodes or *inodes* represent files in the Linux VFS. Each inode is identified by a unique number and contains both the metadata of a file as well as the references to the actual data blocks storing its contents. An overview of the Metadata Stored in an inode of a Linux (or any other POSIX [109] compliant) system is given in Table 5.6. It can be accessed via the `stat` structure returned by the function of the same name in the `libc`. Most notably, the name of the file is not stored in the inode. Nevertheless, regular access to files is not possible using only an inode number.

To actually access any file, a mapping between a file name and an inode is required. This mapping (*hard link* in the Linux terminology) is created by directory entries. Multiple directory entries are collected in directories, each entry linking a file name to an inode via its number. Directories itself are files – as almost everything is a file in Unix systems – and thus directories can contain other directories (named subdirectories). This allows to recursively create a directory tree containing all files stored in the file system. Obviously, if a directory is a file it is also represented by an inode. A minimal, exemplary file system that shows the relationships between directories and files on the on hand and inodes and file contents on the other is depicted in Figure 5.6.

The term hard link is commonly used to denote the possibility to have multiple directory entries linking the same inode. This is the logical next step based on the way

[6] File systems are allowed to deviate internally from these concepts, if their external interface is VFS compatible.

Table 5.6 Metadata of an Inode. This table shows the Metadata Stored in an inode as returned in the `stat` structure by the `stat()` system call [90].

Field	Description
`st_dev`	Device ID of the device containing the inode.
`st_ino`	Inode number.
`st_mode`	Mode of the inode, i.e., access rights and inode file type.
`st_nlink`	Number of hard links to the inode.
`st_uid`	User ID of the inode's owner.
`st_gid`	Group ID of the group owning the inode.
`st_rdev`	Device ID of the represented device in case of a character or block special file.
`st_size`	File size in bytes in case of a regular file or contained pathname's length in case of a symbolic link.
`st_atime`	Time of last data access.
`st_mtime`	Time of last data modification.
`st_ctime`	Time of last inode status change.
`st_blksize`	Preferred block size for file system I/O.
`st_blocks`	Number of 512 byte blocks allocated for this inode.

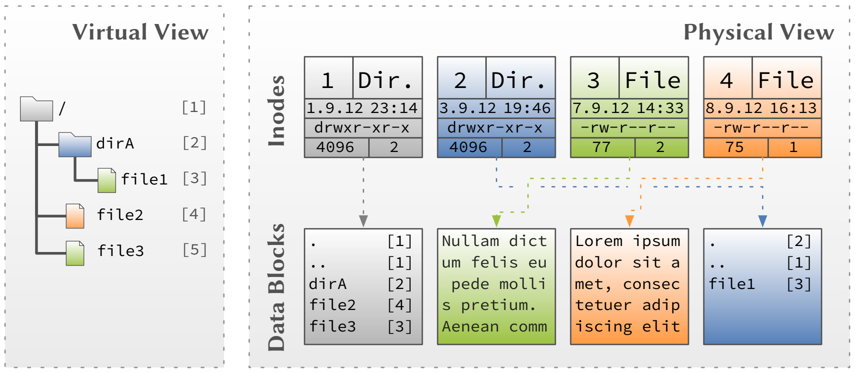


Figure 5.6 Structure of a File System. This figure shows the directory tree (virtual view) and the inodes and data blocks (physical view). The name and corresponding inode number of each directory entry are shown in the tree on the left. Each inode is depicted with a subset of its metadata (number, type, a timestamp, access rights, size and number of hard links) and references to data blocks. Additionally, the figure illustrates the functionality of hard links (`/dirA/file1` and `/file3`) and the fact that directories are just special files.

file systems are built. Creating hard links instead of copying files saves storage space if a file needs to be accessible from multiple parts of the file system. Obviously, when a file is changed via one of those hard links, the change is visible for all other hard links as well. When a file is deleted, the inode is cleared only if there are no further links (directory entries) to it. From the technical point of view, this usage of the term hard link is wrong, because there is no such concept as a primary directory entry and additional hard links and all hard links to a single file are indiscernible except for their file name. The only way for the file system to know whether there are multiple hard links to an inode is the `st_nlink` field of the inode (see Table 5.6). Nevertheless, the term is also used throughout this chapter to denote the case of multiple hard links in absence of a better term.

As there are at least to types of inodes – files and directories – there has to be a way for the file system to differentiate them to be able to work with the directory tree. Each inode has a file type that is encoded in the `st_mode` field of the `stat` structure (see Table 5.6). In total, Linux and POSIX systems support the seven file types shown in Table 5.7.

Table 5.7 Inode File Types. This table shows the seven file types an inode can have, stored in the `st_mode` field of the `stat` structure [90] returned by the system call with the same name.

Type	Description
Directory	A directory, i.e., a special file containing directory entries.
Regular File	A regular file.
Symbolic Link	A link to another file. In contrast to a hard link, the target file is referenced via its name.
Block Device	A device special file representing a block device.
Character Device	A device special file representing a character device.
FIFO	A file representing a named pipe, an inter process communication technique.
Socket	A file representing a local inter process communication socket, also called Unix domain socket.

5.4.1.2 Basic Manifests

A manifest is a tree structure that mirrors the directory tree of a file system and contains the entire metadata of the tree and the files linked therein. Every directory entry contained in the directory tree of the original file system is represented by a node in the manifest that stores the metadata of this particular element[7]. The hierarchy of classes depicted in Figure 5.7 is used to properly represent the directory tree and the corresponding metadata.

The root of the class hierarchy is the *Node* class that stores a node's name, a reference to the node's parent node, as well as the metadata common to all other nodes types. Most fields contained in the `stat` structure (see Table 5.6) of the directory entry's corresponding inode are stored as part of the common metadata. The field `st_dev` identifies the device the inode belongs to and the field `st_blksize` stores the preferred block size for file system I/O. Both values are not required to export an image correctly and thus not stored as part of the manifest. The field `st_rdev` contains an identifier

[7] More precisely, it stores the Metadata Stored in the inode the element is linked to.

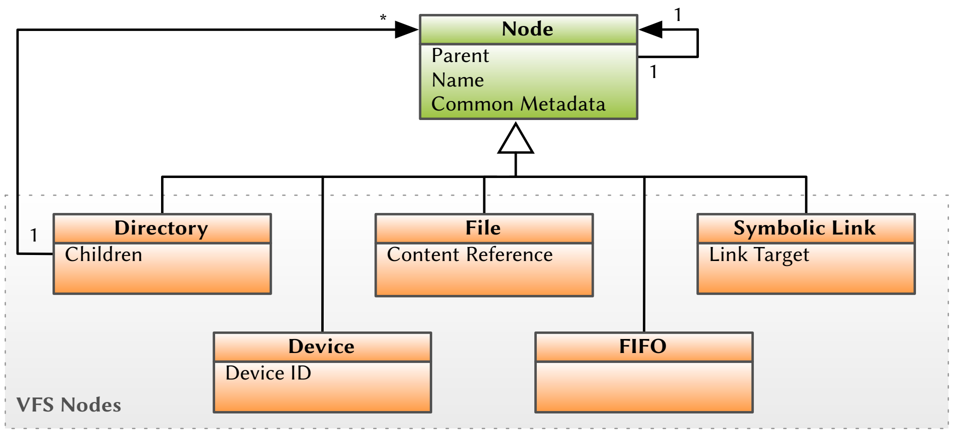


Figure 5.7 The Node Types in a Manifest. This figure shows the different node types used to store the metadata of a file system. Node is the ancestor of all node types. It contains fields to store the name and common metadata from the `stat` structure of the corresponding inode (see Table 5.6). For each of the derived node types additional fields are shown.

of the represented device. It is just valid for block and character device files and thus it is not reasonable to store this identifier for every file.

The remaining classes in the node hierarchy represent the different file types of the Linux VFS. For each but two of the file types listed in Table 5.7 corresponding node types exist. The first exception is the Device node type that represents both block and character devices. Socket files are the second exception. They are not represented in a manifest at all, because they cannot be manually created and thus there is no way to restore a socket file during export. Furthermore, handling socket files in the Image Store is not required, because they are generated implicitly when a Unix socket is created with the `socket()` system call [90].

The most important class is *Directory*, because the nodes of this type form the structure of the tree constituting the manifest. Each *Directory* contains a field *Children* that can in turn contain further nodes of any type. These child nodes represent all directory entries contained in the corresponding directory of the original file system.

The second most important class is *File*. Most of the directory entries in a file system are usually files and thus this is the most frequent node type in manifests. Each *File* stores a reference to the content of the file represented by this node. This reference is central to deduplication: the Data Store returns this reference when either stores a new file or when it has determined that the content is already stored.

A *Symbolic Link* stores the path name of its target, i.e., the path to the file it points to. *Device* nodes represent both character and block device special files. The distinction between those two inode file types is nevertheless possible based on information stored in the *Node* class: the field `st_mode` does not only store access rights, but also a type identifier. Device nodes additionally store the device identifier that is used by the Linux kernel to identify the represented device. Finally, *FIFO* nodes represent named pipes that do not require additional metadata. An exemplary manifest representing the file system shown in Figure 5.6 is depicted in Figure 5.8.

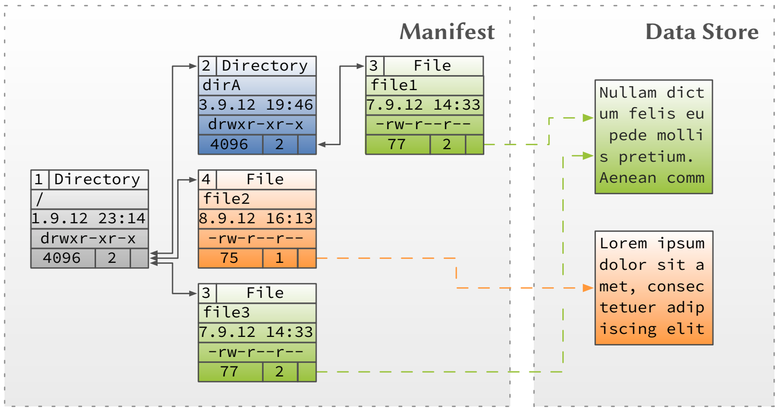


Figure 5.8 Structure of a Manifest. This figure shows the manifest representing the file system shown in Figure 5.6 and Data Store with the content of the two files. Each node is depicted with its type and a link to its parent directory as well as a subset of its metadata (original inode number, one of the timestamps, access rights, size and number of hard links). Directory and File nodes additionally have links to their children and content, respectively.

5.4.1.3 Extended Manifests

Requirement R5.7 explicitly demands support for storing layered virtual machines built using the Image Compositor (see Chapter 4). The composition technique used by the Image Compositor is based on the union mount technology. As already stated in Section 5.3.4.1, union mounts have to use special markers to "delete" files from lower layers if these layers are not writable. There are two types of markers: whiteout files and opaque directory markers. Whiteout files [173, 10, 11, 108] are created in the higher layer to mark the corresponding files (or directories) in the lower layer as deleted and effectively making them inaccessible. In case of aufs a whiteout file named `.wh.sample` in the higher layer marks the file (or directory) named `sample` in the same directory of any lower layer as deleted.

Opaque directory markers [10, 11] are created in the higher layer to prevent merging of the contents of a directory in the higher layer with the contents of corresponding directories in lower layers. Instead, opaque directories completely replace the corresponding directories from lower layers. This kind of marker is required when a directory existing in a lower layer is first deleted – and therefore replaced by a whiteout file – and afterwards a new directory with the same name is created in the higher layer. In case of aufs an opaque directory is marked by adding a file named `.wh..wh..opq` to it.

To fully support layered virtual machines the Image Store needs to correctly represent both whiteout files and opaque directories in the manifest. Two additional node types that do not exist as inode file type in a real file system are thus added to the class hierarchy: the *Whiteout* node and the *Opaque Directory*. The resulting class hierarchy is depicted in Figure 5.9.

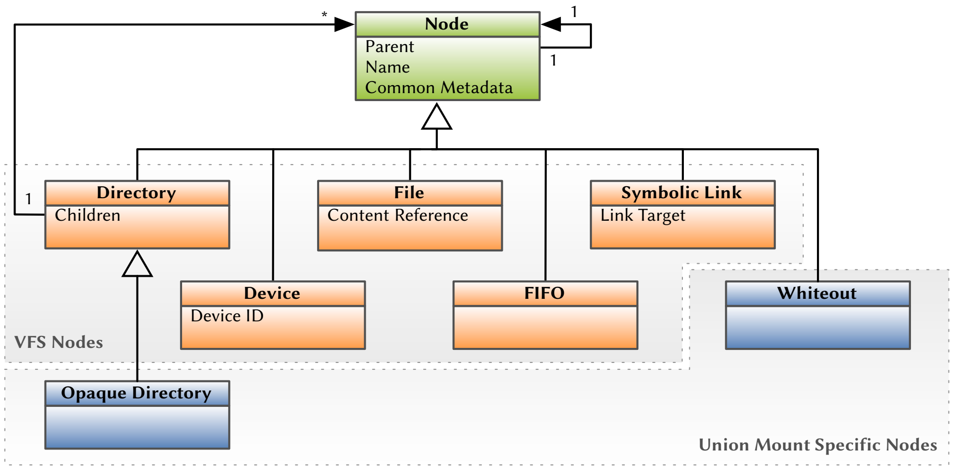


Figure 5.9 The Node Types in a Manifest (Extended). This figure shows the complete class hierarchy of nodes used to store the metadata of a file system. Besides the nodes already included in Figure 5.7 it contains two new node types that represent whiteout files and opaque directories used by union mounts.

A *Whiteout* node represents a whiteout marker, i.e., a file from a lower layer that has been deleted. It does not require any additional metadata, because its sole purpose is the correct interpretation of a whiteout marker during export and for advanced features like the Merge and Diff operations and the analysis of virtual machine images' contents. An *Opaque Directory* represents an opaque directory, i.e., a directory that replaces corresponding directories in lower layers. Note that contrary to the *Whiteout* node it does not represent the marker file itself, e.g., a `.wh..wh..opq` file created by aufs, but a directory with the specific property of being opaque. Thus, the *Opaque Directory* is a subclass of the *Directory*. Similarly to *Whiteout* nodes no additional metadata is required for the same reason.

5.4.1.4 Hard Links in Manifests

Contrary to a Linux file system that has a clear separation between file names on the one side and file metadata and contents on the other side, in the Image Store the boundary is moved: the name of a file and its metadata are combined in the manifest, while the content is stored in the Data Store. This approach simplifies analysis of virtual machine images. Depending on the format used to store the manifest (see Section 5.4.1.6), it allows even manual analysis of manifests. The drawback of this approach is that it does not naturally support multiple hard links to a single file. Without support for hard links the exemplary file system shown in Figure 5.6 will be incorrectly exported to the file system shown in Figure 5.10. Therefore, support for hard links is required for exporting logically equivalent images.

There are two ways how hard links can be represented in a manifest: by introducing a special subclass of *Node* that represents every hard link to a file starting with the

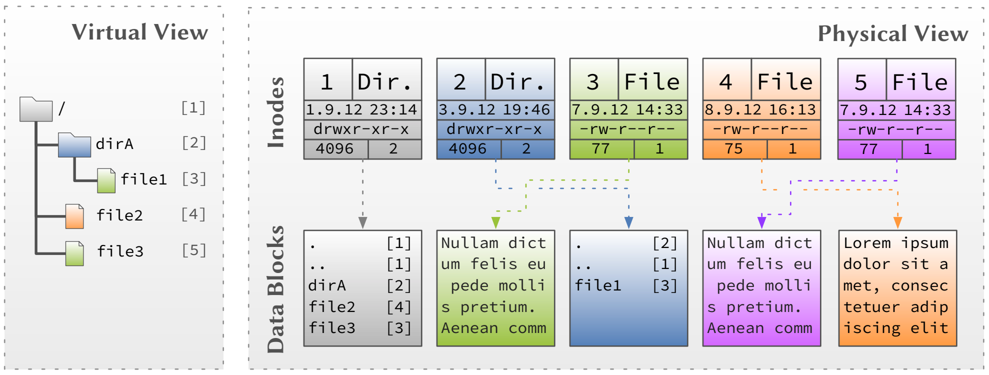

Virtual View
/ [1]
dirA [2]
file1 [3]
file2 [4]
file3 [5]
Physical View
Inodes
1 Dir. 1.9.12 23:14 drwxr-xr-x 4096 2
2 Dir. 3.9.12 19:46 drwxr-xr-x 4096 2
3 File 7.9.12 14:33 -rw-r--r-- 77 1
4 File 8.9.12 16:13 -rw-r--r-- 75 1
5 File 7.9.12 14:33 -rw-r--r-- 77 1
Data Blocks
. [1] .. [1] dirA [2] file2 [4] file3 [3]
Nullam dictum felis eu pede mollis pretium. Aenean comm
. [2] .. [1] file1 [3]
Nullam dictum felis eu pede mollis pretium. Aenean comm
Lorem ipsum dolor sit amet, consectetuer adipiscing elit

Figure 5.10 Export Without Hard Link Support. This figure shows the result of exporting the file system shown in Figure 5.6 after it has been imported into the Image Store, when no support for multiple hard links to a single file exists. In contrast to the original file system `/dirA/file1` and `/file3` are no longer hard links to the same file, but two individual files.

second or by storing an identifier that allows to match corresponding hard links. The former approach requires the hard link detection to be executed during the import process, while the latter approach does the hard link detection during the export. Both approaches have advantages and disadvantages. For easier implementation of the analysis features, the Image Store uses the second approach. The inode number `st_ino` is the natural identifier for matching groups of corresponding hard links and is thus stored for every node in the manifest[8], although the actual number has no meaning for the stored image.

5.4.1.5 In-memory Representation

All node types shown in Figure 5.9 are implemented as Python classes. Because the number of nodes in a manifest is likely very large, special provisions need to be made to reduce footprint of a manifest in memory. Classes in python are typically implemented by dynamic dictionaries that store all fields of a class' instance. On the one hand, this is a very flexible approach, because it enables extending classes with new fields at runtime, and thus it is very well suited for a script language. On the other hand, using a variable length dictionary increases the memory footprint of each instance, because not only the value, but also the name of each field is stored for each single instance[9]. Especially for large numbers of instances, this approach is not very efficient. Another way to implement classes is to explicitly list the fields using the `__slots__` class variable. This fixed list of fields can be stored much more efficiently compared to the dynamic dictionary and thus significantly reduces the footprint of the whole manifest in memory. The list of fields and constructors of the `Node` superclass

[8] Technically it is not necessary to store it for *Directory* nodes, because Linux does not allow hard links to directories. Instead of adding another intermediate superclass to the class hierarchy, the Image Store stores the inode number directly in the *Node*.

[9] Newer versions of Python try to solve this problem by dictionaries sharing keys.

and the directory nodes are shown in Listing 5.1 together with an additional class that encapsulates the common metadata.

```
143 class Stats(object):
144     __slots__ = ['st_ino', 'st_mode', 'st_nlink', 'st_uid', 'st_gid',
145         'st_size', 'st_atime', 'st_mtime', 'st_ctime', 'st_blocks']
146 
147     def __init__(self, stats=None):
148         for key in self.__slots__:
149             setattr(self, key, copy(getattr(stats, key, None)))
  ⋮
231 class Node(object):
232     __slots__ = ['_parent', 'name', 'stats']
233 
234     def __init__(self, name, stats=None):
        ...
242         self.name = name
243         self.stats = stats
244         self._parent = None
    ...
407     def add_to(self, directory):
408         directory._children.append(self)
409         self._parent = directory
  ⋮
548 class Directory(Node):
549     __slots__ = Node.__slots__ + [ '_children' ]
550 
551     def __init__(self, name, stats=None):
552         Node.__init__(self, name, stats)
553         self._children = list()
  ⋮
815 class OpaqueDirectory(Directory):
816     def __init__(self, name, stats=None, directory=None):
817         Directory.__init__(self, name, stats)
818         if directory is not None:
819             for childnode in directory._children:
820                 childnode.add_to(self)
821             directory._children = list()
```

Listing 5.1 Node Class Definitions – Part 1. The list of fields and constructors of the Node superclass and the directory nodes as well as the fields and constructor of a class encapsulating the common metadata.

Instances of the `Stats` class are used to store the metadata of each node in a manifest. The list of fields in the `__slots__` class variable includes most fields of the `stat` structure with the exceptions described above (Lines 144 to 145). The constructor expects the `stat` structure as returned by the `lstat` function of Python's `os` module as its only argument. Using Python's metaprogramming techniques it copies the values from the `stat` structure into its corresponding instance variables (Lines 148 to 149).

The `Node` class is a straightforward implementation of the root class. It stores a pointer to its parent node, i.e., its ancestor in the tree of nodes, in the `_parent` field[10]. Regarding the file system the manifest represents, the ancestor is the directory that contains a directory entry. Furthermore it stores the name of the directory entry in

[10] The reason why the field name starts with an underscore letter is explained in Section 5.4.4.2.

`name` and the metadata of the represented inode in a `Stats` instance stored in `stats` (Line 232).

The constructor of the `Node` class only requires the name of the directory entry as argument, whereas the `Stats` instance containing the corresponding inode's metadata is an optional argument. This approach simplifies the code required to reconstruct nodes from a manifest stored on disk using a generic function. The name and metadata are then stored after some sanity checks (not shown) in Lines 242 to 244. Note that the node's ancestor is initialized with `None`, i.e., the node is not yet part of the tree of nodes. The `add_to` method is used to set the parent node and register a child node with its parent (Lines 407 to 409).

The `Directory` class extends the list of fields with the `_children` field that contains a list of child nodes. This list is initialized as an empty list (Line 553), i.e., the directory has no child nodes just as a node has no parent after creation. Note that the `add_to` method of the `Node` class (Lines 407 to 409) does not use a special method of the `Directory` class to add a child to it, but directly appends the node to the `_children` list without checking for existing entries with the same name. This approach is suitable for manifests, because they represent existing file systems that do not allow two directory entries with the same name in a single directory. Consequently, no collisions can occur in a manifest.

The `OpaqueDirectory` class is the only node type that is not derived directly from `Node`, but from `Directory` (Line 815). Note that its constructor accepts an existing `Directory` instance as optional argument. If another directory is pass, all of its children are added to the newly created `OpaqueDirectory` instance (Lines 819 to 820). It is important to remove the children from their original `Directory` instance by assigning an empty list to its `_children` field (Line 821), because the `add_to` method does not take care of removing a node from its old parent node if it is registered with a new one.

Listing 5.2 shows the list of fields and constructors, if any, of the classes representing the remaining node types: files, symbolic links, device files, named pipes and whiteout files.

The `SymbolicLink` and `Device` classes both extend the list of fields with an additional field: `target` for storing the symbolic link's target and `rdev` for storing the ID of the represented device, respectively. Both classes' constructors accept values for these fields as optional arguments. The `FIFO` class on the other hand has no additional fields and thus does not need a special constructor.

The `File` class extends the list of fields with the `hash` field that stores the content reference: a hash value of the file that is used to identify a file in the Data Store Section 5.4.2.2. Contrary to the `SymbolicLink` and `Device` classes, the constructor of `File` does no accept an optional argument to pass the hash value, because it is typically not known when the `File` instance is created, but instead initializes the field with `None` (Line 844).

Like the `FIFO` class, the `Whiteout` class has no additional fields and thus does not need a special constructor.

```
class SymbolicLink(Node):
    __slots__ = Node.__slots__ + [ 'target' ]

    def __init__(self, name, stats=None, target=None):
        Node.__init__(self, name, stats)
        self.target = target
class Device(Node):
    __slots__ = Node.__slots__ + ['rdev']

    def __init__(self, name, stats=None, rdev=None):
        Node.__init__(self, name, stats)
        self.rdev = rdev
class FIFO(Node):
    ...
class File(Node):
    __slots__ = Node.__slots__ + [ 'hash' ]

    def __init__(self, name, stats=None):
        Node.__init__(self, name, stats)
        self.hash = None
class Whiteout(Node):
    ...
```

Listing 5.2 Node Class Definitions – Part 2. The list of fields and constructors, if any, of the classes representing the different node types (except directories).

To create a complete manifest from a tree of nodes a class of the same name is used (not shown). It contains a reference to the root of the node tree and metadata about the manifest itself: an UUID and a version number to uniquely identify a manifest, information about the original file system, e.g., the file system type, as well as information about the version history of the manifest, e.g., whether it was cloned from a specific manifest or is the result of a Merge operation between to specific manifests.

5.4.1.6 On-disk Representation

Manifest can be stored on disk in two different formats. The first format is an *Extensible Markup Language* (XML) document. An XML document is the natural storage format for manifests, because a manifest is basically a tree of nodes. The biggest advantage of the XML format is that it is human-readable. This allows manual inspection of manifests and simplifies the development of the Image Store itself. The second format is a serialized version of the manifest including the node tree. For efficient serialization of the manifest Python's *pickle* library is used. The resulting file is not human readable, but much smaller and it can be read significantly faster. The XML format can optionally be *pretty printed*, i.e., outputted with indentation to improve readability, and both formats can be compressed using zlib to reduce the storage requirements of the manifests, which is especially important for the XML format.

An exemplary excerpt from a manifest in the pretty printed XML format is shown below.

```xml
<manifest uuid="6d6aab3a-53a3-4282-b423-73f63adfe297" version="1">
  <root>
    <node name="/" type="Directory">
      <stats>(...)</stats>
      <node name="lib64" type="SymbolicLink">
        <stats>(...)</stats>
        <target type="str">/lib</target>
      </node>
      <node name="lib" type="Directory">
        <stats>(...)</stats>
        <node name="libnss_dns.so.2" type="SymbolicLink">
          <stats>
            <st_ino type="int">221242</st_ino> (...)
            <st_nlink type="int">1</st_nlink> (...)
          </stats>
          <target type="str">libnss_dns-2.11.3.so</target>
        </node> (...)
        <node name="libnss_dns-2.11.3.so" type="File">
          <stats>
            <st_ino type="int">221213</st_ino> (...)
            <st_nlink type="int">1</st_nlink> (...)
            <st_size type="int">22928</st_size> (...)
          </stats>
          <hash type="str">ef1330668abf9ca82622832a1321e93560894764</hash>
        </node> (...)
      </node> (...)
      <node name="bin" type="Directory">
        <stats>(...)</stats>
        <node name="gunzip" type="File">
          <stats>
            <st_ino type="int">40974</st_ino> (...)
            <st_nlink type="int">2</st_nlink> (...)
            <st_size type="int">63</st_size> (...)
          </stats>
          <hash type="str">3f8ef0b538cb322fc4220edd45a21171630507c6</hash>
        </node>
        <node name="uncompress" type="File">
          <stats>
            <st_ino type="int">40974</st_ino> (...)
            <st_nlink type="int">2</st_nlink> (...)
            <st_size type="int">63</st_size> (...)
          </stats>
          <hash type="str">3f8ef0b538cb322fc4220edd45a21171630507c6</hash>
        </node> (...)
      </node> (...)
    </node>
  </root>
  (...)
</manifest>
```

Exemplary excerpt from a manifest in the pretty printed XML format.

The excerpt focuses on the tree of nodes and does not show most of the manifest metadata except its UUID and version. It shows the root directory of the file system, the `/lib` directory containing the `libnss_dns-2.11.3.so` library and the `libnss_dns.so.2` symbolic link to it, the `/lib64` symbolic link to `lib`, as well as the

`/bin` directory containing the `gunzip` and `uncompress` binaries that are hard links to the same file.

Note that the XML format is containing a lot of redundant information, e.g., the `type` attributes in the `<st_...>` and `<hash>` tags. These attributes enable the use of a generic function to parse the XML and recreate the in-memory representation of the manifest. This is useful for the development process, but an optimization of the format with specific parsing code might improve the parsing time.

5.4.2 Storage Architecture

This section describes the architecture of the storage system used by the Image Store that is comprised of Metadata Store and data Store.

5.4.2.1 Metadata Store

Manifests are stored as plain files in the Metadata Store. This approach has been chosen because it simplifies development by allowing manual inspections of the Metadata Store and the manifests it contains using standard tools that could not be used if the manifests were stored in a database. The file name of a manifest is created by concatenating the UUID of the manifest with its version number (separated by a dot). The exemplary manifest shown above would be stored in a file named:

```
6d6aab3a-53a3-4282-b423-73f63adfe297.1
```

The Metadata Store ensures that the older versions of a virtual machine are preserved simple by increasing the version number of the manifest if a virtual machine is updated. This ensures that no manifest is ever overwritten. Note that there is no way for the Image Store to determine that an imported image is an update of an already existing image. This information has to be supplied by the user during the import process. Otherwise the resulting manifest will be assigned a different UUID and is treated like a new virtual machine image. The `mismgr` refuses import an image without either specifying a previous version or explicitly marking the image as new.

The correlation between the manifests of different versions of a virtual machine image is immediately visible when manifests are stored in the way described above. Additionally, the latest version of an image can be looked up easily by searching for the manifest with the correct UUID and the highest version number in its file name.

To check whether a virtual machine image has been cloned or created from scratch, the version history information in the manifest of this image's first version has to be checked. If it is a cloned image, the source image and version is recorded there. Note that the information regarding clones is only recorded for cloned images, but not for the source images. This is due to the requirement to never change a manifest once it is stored in the Metadata Store. To determine all clones of a given image, the version history information in the manifests of the first version of every stored image need to be checked or an external list of clones needs to be maintained.

5.4.2.2 Data Store

The contents of files are stored in the Data Store. It implements a content-addressable storage to provide deduplication on the granularity of entire files and uses exchangeable back ends allow to further improve the storage efficiency. The contents are stored as individual files in a regular file system.

For referencing content in the Data Store, a hash function is applied at the content and the resulting hash value is used as address. The Image Store uses the *SHA-1* cryptographic hash function [50] for calculating hash values. To reduce the possibility of collisions the length of the file is appended to the hash value, as described by Wang et al. [164]. This does not only help to prevent accidental collisions, but also collision attacks targeted at gaining access to a file stored in the Image Store by inserting a file with the same SHA-1 value into an image, import the image and export it again.

To limit the number of files generated in a single directory, the first two characters of the hash value are removed from the file name and used as directory name. This approach is also used by the *git* version control system and it proved to be effective. The complete path name generation process is shown in Figure 5.11.

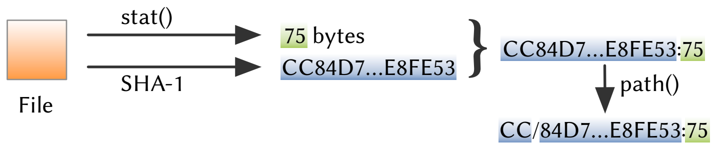


Figure 5.11 Path Name Generation. The generation of a path name for the Data Store. First, the size of the file is concatenated with the hash value. Then, the first two characters are removed from the name to form the directory name

Lines 57 to 69 in Listing 5.3 show the methods of the `Datastore` class that implement the path name generation. The code is split into multiple methods because different parts of the generated path are required in different parts of the Data Store implementation. The `generatePathAndName` method separates the hash in the directory and file path (Line 66), whereas the `generatePath` method only returns the directory part (Line 69). The `getPath` and `getPath2` methods both return a complete path including the path to the Data Store and the length of the file (Line 63), although the former just calls the latter for the actual path name generation (Line 60).

Files are stored in the Data Store using the `saveData` method (Lines 71 to 90). First, the method calculates the hash value for the file (Line 79), if it is not stored in the node yet, and generates the final path for the file's content (Line 81). Then, it checks whether the content is already stored there and returns immediately in this case (Lines 84 to 85). This is where the actual deduplication takes place. If the content is new, it is copied into the Data Store using the `store` method of the back end (Line 90) after creating the selected directory if necessary (Lines 87 to 88).

The Data Store is equipped with different back ends. Two of them are shown in Listing 5.4: the *default* and the *gzip* back end. All back ends are derived from the `Backend` class that implements a few fundamental methods for Data Store maintenance (not shown).

```python
class Datastore(object):
    ...
    def getPath(self, node):
        filehash = node.hash
        filesize = str(node.stats.st_size)
        return self.getPath2(filehash, filesize)

    def getPath2(self, filehash, filesize):
        return self.path + self.generatePathAndName(filehash) + ':' +
     filesize

    def generatePathAndName(self, filehash):
        return filehash[0:2] + '/' + filehash[2:]

    def generatePath(self, filehash):
        return filehash[0:2]

    def saveData(self, node, srcpath):
        ...
        if not node.hash:
            node.hash = self.hash(srcpath)

        dstdir  = self.path + self.generatePath(node.hash)
        dstpath = self.getPath(node)

        if os.path.exists(dstpath):
            return

        if (not os.path.exists(dstdir)):
            os.makedirs(dstdir)

        self.backend.store(srcpath, dstpath, node.stats.st_size)

    def copyData(self, exporter, node):
        self.backend.retrieve(self.getPath(node), exporter.getPath(node),
            node.stats.st_size)

    def openData(self, node):
        return self.backend.openForReading(self.getPath(node))
```

Listing 5.3 Data Store Implementation. This listing contains important parts of the implementation of the Data Store: the generation of path names that is vital to the deduplication and the functions to store file in the Data Store, to open stored files, and to retrieve files from the Data Store.

Every back end must implement four methods that are used both by the `Datastore` and the `Backend` classes. The `openForReading` method opens a file stored in the Data Store for reading and returns a *File Object* that can be used to read content of the file. The `decompressObject` returns an instance of the library used to compress contents in the Data Store. This instance is used in one of the methods in the `Backend` class that accesses the contents directly to decompress them. If no compression is used, this method must return `None`. Finally, the `retrieve` and `store` methods copy a file out of and into the Data Store.

The *default* back end (`DefaultBackend`) stores the file contents unmodified. Therefore the implementation is trivial. Files can be opened for reading and writing using

```
142 class Backend(object):
 ⋮      ...
177 class DefaultBackend(Backend):
178     def openForReading(self, src):
179         return open(src, 'rb')
180 
181     def decompressObject(self, size):
182         return None
183 
184     def retrieve(self, srcpath, dstpath, size):
185         self.copy(open(srcpath, 'rb'), open(dstpath,'wb'))
186 
187     def store(self, srcpath, dstpath, size):
188         self.copy(open(srcpath, 'rb'), open(dstpath,'wb'))
 ⋮
210 class CompressingBackend(Backend):
211     def __init__(self, level, blocksize=4096):
212         Backend.__init__(self)
213         self.level = level
214         self.blocksize = blocksize
215 
216     def decompressObject(self, size):
217         return self._decompressObject()
218 
219     def retrieve(self, srcpath, dstpath, size):
220         srcfile = self.openForReading(srcpath)
221         self.copy(srcfile, open(dstpath, 'wb'))
222 
223     def store(self, srcpath, dstpath, size):
224         dstfile = self.openForWriting(dstpath)
225         self.copy(open(srcpath, 'rb'), dstfile)
226 
227 class ZlibBackend(CompressingBackend):
228     def openForReading(self, srcpath):
229         return gzip.open(srcpath, 'rb')
230 
231     def openForWriting(self, dstpath):
232         return gzip.open(dstpath, 'wb', compresslevel=self.level)
233 
234     def _decompressObject(self):
235         return zlib.decompressobj(16 + zlib.MAX_WBITS)
```

Listing 5.4 Data Store Implementation – Back Ends. This listing contains the implementation of the *default* and *zlib* back end.

the standard `open` function of python (Lines 179, 185 and 188). The implementation of the `decompressObject` method just returns `None` to signal that no compression is used.

The modular design of the Image Store makes it possible to quickly write new back ends. A few additional back ends are provided that leverage different techniques to further reduce the size of the Data Store after the deduplication is applied. The first back end (not shown) detects holes in the content, i.e., blocks containing only zeros, and stores the content as sparse file in the Data Store by skipping those holes using the `seek` system call. During export, these files are restored as regular files, except if they had already been sparse files before the import.

Further back ends exist that use different compression libraries to compress the contents before storing them. The content of every file is individually compressed by those back ends during the import process and decompressed during the export process. Each of those back ends is configurable with regard to the compression level that is typically a value between 0 and 9. The libraries used by the back ends are zlib, the library used by gzip, libbzip2, the library used by bzip2, and liblzma, the library used by xz. In the remainder of this thesis these back ends are called *zlib*, *bzip*, and *lzma* for brevity. Whenever a specific combination of back end and compression level should be denoted the name of the back end is concatenated with the compression level, e.g., *zlib1* for the zlib backend with compression level 1.

The class `CompressingBackend` is the superclass of all compressing back ends. It already implements the `decompressObject`, `retrieve`, and `store` methods in a generic way (Lines 216 to 225). The actual back ends just needs to implement two more methods: `openForWriting`, which is the counterpart to `openForReading`, and `_decompressObject`, which returns an appropriate library instance that can be used to decompress file contents. Additionally, the class provides a generic constructor that is sufficient for all provided compressing back ends.

The actual back end using the zlib library is simple to implement using the generic superclass. It solely consists of the two methods that open a file for reading and writing using the open function of the zlib library[11] (Lines 228 and 232) and the function that returns a decompress object (Line 235). The back ends using the libbzip2 and liblzma libraries (not shown) are equally simple to implement.

The Data Store also provides an *integrity check* that is done by calculating the proper path of every file in the Data Store using the process show in Figure 5.11 and comparing it with the actual path of the file. This requires the file to be decompressed or reconstructed if a compression back end or the sparse back end is used, respectively. This operation works solely on the Data Store and does not require any information from the Metadata Store. Using the integrity check errors in individual files stored in the Data store can be detected that would have remained undetected in traditional images.

5.4.3 Import and Export of Images

The import and export of virtual machine images are central processes of the Image Store. In this section, the implementation of both processes is described. Additionally, an approach is shown that enables modifying files on the fly during the export process and can be used to make minor changes during the deployment of a virtual machine.

5.4.3.1 Import

The import process is responsible for extracting the metadata and content of all files from a virtual machine image, encapsulating the metadata in a manifest, and storing

[11] There are two relevant libraries in Python: the `gzip` library provides an way to transparently work with compressed files, whereas the `zlib` library allows to explicitly compress or decompress buffers.

the contents in the Data Store. The structure of manifests has already been described in Section 5.4.1 and the Data Store has been described in Section 5.4.2.2. The missing part is the creation of a manifest from a virtual machine image.

Listing 5.5 shows the `searchFiles` function that is responsible for building the tree of nodes that constitutes the main part of the manifest. It delegates this task to the `_searchFiles` function that is called recursively for each directory entry in the virtual machine image to be imported. It expects the arguments listed in Table 5.8.

Table 5.8 Arguments to the `_searchFiles` Function. This table lists the arguments of the recursive `_searchFiles` function and describes their meaning for the import process.

Argument	Description
`datastore`	An instance of the `Datastore` class that provides access to the Data Store. Might be `None` if only the manifest should be created. Passed unmodified when the function calls itself recursively.
`calcHash`	A Boolean variable that controls whether hash values are calculated for files when no Data Store is used. This is used in the *Diff-based Reimport process* described in Section 5.4.6.1. Passed unmodified when the function calls itself recursively.
`src`	The path to the mounted virtual machine image. Passed unmodified when the function calls itself recursively.
`subpath`	The path of the directory entry to process in the current call. This path is relative to `src`. Controls the directory entry processed in the current call.
`name`	The name of the directory entry to process in the current call. This is identical to `os.path.basename(subpath)`. Controls the directory entry processed in the current call.

The function `_searchFiles` starts by computing the absolute path of the directory entry to be processed in the current call (Line 106). For the initial call in the import process (Line 103) the computed path is `src + ''` and thus the mount point of the virtual machine image, i.e., its root directory. Afterwards, the `stats` structure of the directory entry is retrieved and the common metadata is extracted and stored in an instance of the `Stats` class (Lines 107 to 108). Based on the inode file type a node of the corresponding type is created and returned in Lines 110 to 147.

The simplest case is a named pipe that consists of nothing more than a name (passed as argument) and the common metadata (retrieved from the `stats` structure. An instance of the `FIFO` class is creating with this information to represent the current directory entry and returned to the caller (Lines 144 to 145).

Device files and symbolic links as almost as simple to handle as named pipes. For the former, in addition to the name and common metadata the device ID of the represented device needs to be passed as an additional argument to the `Device` constructor (Lines 141 to 142). It does not matter if the device file represents a block or character device, because this information is already stored in the `st_mode` field of the `Stats` instance. In the case of a symbolic link the link's target needs to be passed

```python
def searchFiles(datastore, calcHash, src):
    return _searchFiles(datastore, calcHash, src, '', '')

def _searchFiles(datastore, calcHash, src, subpath, name):
    path = src + subpath
    orig_stats = os.lstat(path)
    stats = Stats(orig_stats)

    if stat.S_ISLNK(stats.st_mode):
        return SymbolicLink(name, stats, os.readlink(path))

    if stat.S_ISREG(stats.st_mode):
        if name.startswith(".wh."):
            return Whiteout(name[4:], stats)

        node = File(name, stats)
        if datastore is None:
            if calcHash:
                node.hash = fileops.hash(path)
        else:
            datastore.saveData(node, path)
        return node

    if stat.S_ISDIR(stats.st_mode):
        node = Directory(name, stats)
        for childname in os.listdir(path):
            if (childname == '.wh..wh.orph') or (childname == '.wh..wh.plnk'):
                node.stats.st_nlink -= 1
                continue
            if (childname == '.wh..wh..opq'):
                node = OpaqueDirectory(name, stats, node)
                continue

            childpath = subpath + '/' + childname
            child = _searchFiles(datastore, calcHash, src, childpath, childname)
            child.add_to(node)

        return node

    if stat.S_ISBLK(stats.st_mode) or stat.S_ISCHR(stats.st_mode):
        return Device(name, stats, orig_stats.st_rdev)

    if stat.S_ISFIFO(stats.st_mode):
        return FIFO(name, stats)

    return NullNode()
```

Listing 5.5 Virtual Machine Image Import. The `searchFiles` function shown in this Listing is responsible for extracting the metadata from a mounted virtual machine image, building the tree of nodes that is the central part of a manifest, and copy the files' contents in the Data Store.

as additional argument to the `SymbolicLink` constructor. The link's target can be retrieved using the readlink function in the os module (Lines 110 to 112). In both cases the resulting node is returned to the caller immediately.

Special care has to be taken when current directory entry points to regular files (Lines 113 to 123), because layered virtual machine images contain whiteout files that are on the one hand just plain files, but on the other hand require special treatment so that the Merge and Diff operations work as expected. The current version of the Image Store recognizes only whiteout files created by aufs, although it can easily be extended to support other union mount implementations as well. Whiteout files are files named “.wh.<name>” in aufs, whereby <name> is the name of the file that should be marked as deleted. The code in Lines 114 to 115 recognizes whiteout files and returns Whiteout instance with the name and common metadata to the caller. Note that the “.wh.” part is stripped from the name to achieve a union mount independent representation of the whiteout file.

For regular files that are no whiteout files, an instance of the File class is created using the name and common metadata (Line 117). The further steps depend on whether a Data Store was passed to the function or not. If a Data Store was passed, the file’s content is stored in it (Line 122). In this process the hash value of the file is calculated and stored in the hash field of the File instance. If no Data Store was passed, it depends on the value of the calcHash argument whether the hash value of the file is calculated or not (Lines 119 to 112). Finally, the resulting node is returned to the caller.

Directories are the most sophisticated file type the _searchFiles function has to process. First, an instance of the Directory class is created using the name and common metadata (Line 126). Then, the function loops over all directory entries contained in this directory, which are retrieved using the listdir function in the os module (Line 127). The code in Lines 135 to 137 is executed for every directory entry that is not one of the three special cases that are explained below. First, this code computes the path of the directory entry (relative to src) by combining the path of the directory itself (subpath) with the name of the directory entry (Line 135). Then, the function calls itself recursively passing the first three unmodified arguments together with the path of the directory entry and its name (Line 136). Finally, the node returned by this recursive call that represents the directory entry is then added to the Directory instance (Line 137). When the loop is finished, the directory has been processed completely and the Directory instance is returned to the caller (Line 139).

As stated above, there are three special cases related to aufs that have to be addressed when processing directories. The first two cases are two directories used by aufs at runtime: “.wh..wh.orph” and “.wh..wh.plnk”. Their contents are meaningful only for the union mount they were created in. Consequently, storing them in the manifest is unreasonable. They are omitted by continuing the loop with the next directory entry (Line 130). Note that omitting a directory changes the number of hard links (st_nlink) of the parent directory, because one “..” hard link is missing. Therefore, the number is decreased by one (Line 129).

Opaque directory markers are the third case. In aufs, opaque directories are marked by adding a file named “.wh..wh..opq” to them. This file can appear anywhere in the list of directory entries, so it is not possible to create an OpaqueDirectory instance instead of the normal `Directory` instance in the first place (Line 126) without looping through the directory entries twice. Instead, the special functionality of the `OpaqueDirectory` constructor is used by passing the `Directory` instance to together with the name and common metadata. The resulting instance is the parent of all nodes that were children of the `Directory` instance and replaces it (Line 132). The loop is then continued with the next directory entry (Line 133).

At the end of the importing process, the initial call to `_searchFiles` returns the complete tree of nodes for the virtual machine image to the `searchFiles` function that in turn returns it to its caller. The caller then embeds the tree into an instance of the `Manifest` class as described in Section 5.4.1.5. Thereby, the import process of the virtual machine image is finished.

5.4.3.2 Export

An virtual machine image stored in the Image Store is exported by creating an empty disk image and recreating the files and directories represented by the manifest's tree of nodes in this disk image. The basic approach is to create directories, copy the contents of files from the Data Store to restore files, and use specific system functions to recreate special files. After directories, files, or special files have been created, it is important to apply the metadata recorded in the manifest. If this step is omitted, all exported files and directories have the user and group IDs of the Image Store process as well as the wrong access rights and timestamps, which is seriously affecting the *Discretionary Access Control* implemented in Linux system.

As already stated in Section 5.4.1.4, a manifest cannot naturally support multiple hard links to a single file, i.e., regular files, special files, and symbolic links. Consequently, hard link detection has to be employed in the export process to ensure that hard links are correctly exported. The detection algorithm depicted in Figure 5.12 depends on the values of the `st_ino` and `st_nlink` values stored in the `Stats` instance of each node.

The idea of the algorithm is to maintain a list of the `st_ino` values of all nodes except directories with a `st_nlink` value greater than one that have already been exported. Note that the algorithm uses the values of the node's `Stats` instance and not the actual values of the exported file. The list contains not only the `st_ino` values, but also the path of the exported file.

Whenever a node with a `st_nlink` value greater than one is to be exported, the algorithm checks whether the node's `st_ino` value is already in that list. In this case, the path of the exported file is retrieved from the list and a link to that file is created for the node instead of exporting it again. Otherwise, the `st_ino` number is added to the list together with the path of the exported file and the node is exported as usual.

Unfortunately, not all of the metadata recorded in the nodes' `Stats` instances can be explicitly applied to exported files. Table 5.9 lists these fields and describes why they cannot be applied. With this information the resulting differences between the original virtual machine image and an exported version can be considered. The critical fields are `st_ino`, `st_ctime`, and `st_blocks`. The first and the last field are less

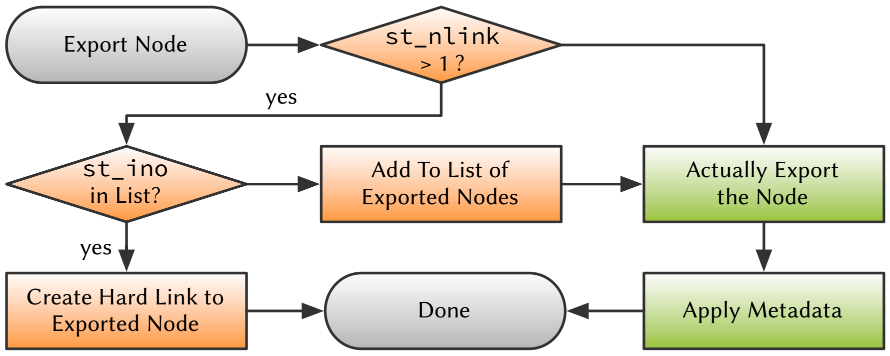


Figure 5.12 Hard Link Aware Exporting. This flow chart describes the process of exporting a node that is correctly handling hard links by only exporting the first node of a group of nodes with identical `st_ino` numbers that represent a single file, symbolic link, special file, or named pipe. (orange part of the flow chart). The regular export process for nodes with only a single hard link is depicted by the green parts of the chart. Note that this process does not apply to directory nodes.

critical, because most applications don't care about inode numbers or the number of allocated blocks. However, the inode change time is an important, visible attribute of a file. For example, backup programs use it as a reliable file modification timestamp, particularly because it is the only timestamp of a file that cannot be changed by a user.

There are workarounds to modify the inode change time: GNU stroke [56] modifies the system time before it writes a file. This approach is applicable for individual files, but not for the large numbers of inodes that have to be created during the export of a virtual machine image[12]. Additionally, this method is not very precise. If another process is scheduled between updating the system time and changing the inode, the approach cannot guarantee that the right timestamp is written to the inode. The only way to reliably set the inode change time is to access the raw image and write the timestamps manually to the correct location. This approach is specific to individual file system types, requires in-depth knowledge of the respective file system implementation and is generally not advisable.

An exported virtual machine image is not logically equivalent to the original virtual machine image, because of the inability to set the inode change time. Fortunately, except for backup programs there are almost no applications that rely on correct inode change times in exported virtual machine images. Even the Image Store itself has only a single feature that relies on correct inode change times: the Diff-based Reimport process described in Section 5.4.6.1. For this specific feature a workaround exists that can deal with the differences in the inode change time between exported image and manifest. For most applications and the particularly purpose of the Image Store the exported image can thus be considered logically equivalent to the original image.

Listing 5.6 shows both generic export methods in the `Stats` and `Node` classes that are used by all other node types as well as specific export methods for symbolic links,

[12] A basic installation of Debian/GNU Linux contains more than 15,000 files.

Table 5.9 Non-exportable Metadata Fields in `Stats`. This table lists the fields of the `Stats` class that cannot be explicitly applied to exported files for different reasons.

Argument	Description
`st_ino`	When a file or directory is exported, the target file system automatically selects a free inode from it pool of free inodes. There is no way to influence the inode selection.
`st_mode`	The field `st_mode` contains two distinct parts: the access rights and the inode file type. The latter is set automatically during the creation of the file or directory depending on the method used, e.g., the `mkdir`, `mknod`, `mkfifo`, or `symlink` functions of the `os` module or the plain open function. Only the access rights part can be applied to exported files or directories. However, after a successful export the inode file type should match the recorded type.
`st_nlink`	The number of hard links to an inode is maintained by the file system. The only way to influence is by creating or removing hard nodes. However, after exporting the entire manifest the number should be matching the one recorded in the manifest.
`st_size`	The size of a file is automatically set after its content is written. It should match the size recorded in the manifest if the export was successful.
`st_ctime`	The time of the last inode status change cannot be set manually, because it is automatically updated with the current time whenever the in the inode is changed.
`st_blocks`	The number of blocks allocated for a file is automatically set after its content is written. It should match the number of blocks recorded in the manifest if the export was successful, but it can differ even after a successful export, e.g. when the export code tries to detect holes in files and exports the file as a sparse file.

device files and named pipes. Listing 5.7 shows the specific export methods for the remaining node types: directories, opaque directories, files, and whiteout files.

The `export` method of the `Stats` class is responsible for applying the stored metadata to exported nodes. Contrary to all other export methods it expects the path of the exported file as argument. The method distinguishes between symbolic links on the one hand and all other inode file types on the other hand. Note that the `Stats` instance does not store a reference to the corresponding node. Thus, the type has to be determined using the value stored in the `st_mode` field (Line 221). For everything except symbolic links, the metadata is applied by changing the exported file's user and group IDs, setting its access rights, and updating its time of last access and last modification in Lines 225, 226 and 227, respectively.

Symbolic Links are special in multiple ways. First of all, the functions used to apply the metadata (`chown`, `chmod`, and `utime` in the `os` module) affect the symbolic links target instead of the symbolic link itself. For the `chown` function a counterpart named `lchown` exists. No counterpart exists for `utime`. The function `lutime` called in Line 223 is provided in a custom module implemented in the *C* programming language

```
class Stats(object):
    ...
    def export(self, path):
        if stat.S_ISLNK(self.st_mode):
            os.lchown(path, self.st_uid, self.st_gid)
            os.lutime(path, (self.st_atime, self.st_mtime))
        else:
            os.chown(path, self.st_uid, self.st_gid)
            os.chmod(path, self.st_mode)
            os.utime(path, (self.st_atime, self.st_mtime))
⋮
class Node(object):
    ...
    def export(self, datastore, exporter):
        fileops.remove(exporter.getPath(self))
        if (self.stats.st_nlink > 1):
            if (self.stats.st_ino in exporter.linkcache):
                os.link(exporter.linkcache[self.stats.st_ino], exporter.getPath(self))
            else:
                path = self.do_export(datastore, exporter)
                if path is not None:
                    exporter.linkcache[self.stats.st_ino] = path
        else:
            self.do_export(datastore, exporter)
⋮
class SymbolicLink(Node):
    ...
    def do_export(self, datastore, exporter):
        path = exporter.getPath(self)
        os.symlink(self.target, path)
        self.stats.export(path)
        return path
⋮
class Device(Node):
    ...
    def do_export(self, datastore, exporter):
        path = exporter.getPath(self)
        os.mknod(path, self.stats.st_mode, self.rdev)
        self.stats.export(path)
        return path
⋮
class FIFO(Node):
    def do_export(self, datastore, exporter):
        path = exporter.getPath(self)
        os.mkfifo(path)
        self.stats.export(path)
        return path
```

Listing 5.6 Virtual Machine Image Export – Part 1. This listing shows the generic export methods in the `Stats` and `Node` classes and the specific export methods for symbolic links, device files and named pipes.

and registered in the `os` module[13]. Note that symbolic links always have the access rights `rwxrwxrwx` or 0777. Consequently, no `lchmod` function exists.

The `export` method of the `Node` class is central to the export process, because it implements the hard link detection algorithm shown in Figure 5.12. It expects two

[13] In recent versions of python all three functions are extended with a `follow_symlinks` keyword argument that facilitates modifying the symbolic links instead of their targets.

arguments: a reference to the Data Store and an instance of the `Exporter` class shown in Listing 5.9 that is used to calculate absolute paths for exporting nodes, store the list of already exported nodes for hard link detection and provide support for exporting whiteout files and opaque directories for specific union mount implementations.

Before a node can be exported, the Image Store must ensure that nothing else is stored at the export path of that node. While this is the case in general, because the export process uses blank image files as target for exporting, Section 5.4.6.2 presents a use case for exporting to existing image files. The `remove` function (shown in Listing 5.8) from the `fileops` module is used for this task.

The list of exported nodes used in the hard link detection algorithm shown in Figure 5.12 is implemented using a *dictionary* named `linkcache` in the `Exporter` instance. The `st_ino` values are the keys and the export paths are the values. Line 396 corresponds to the green part of the algorithm: if a node has only a single hard link, the node type's implementation of the `do_export` method is called to export the node. Lines 389 to 394 correspond to the orange part of the algorithm: if the node's `st_ino` value is already contained in the list of exported nodes, a hard link to the exported node is created (Line 390). Otherwise, the node is exported as usual using the `do_export` method (Line 392), which is required to return the path to the exported node. The path is then stored in the list of exported nodes using the node's `st_ino` value as key (Line 394).

The `do_export` methods for symbolic links, device files and named pipes are almost identical except for the specific function of the `os` module used to export the node in Lines 503, 528 and 540, respectively. Afterwards, the metadata is applied using the `export` methods of the node's `Stats` instance and the path of the exported node is returned.

The `Directory` class has to overwrite the `export` method defined in its superclass `Node`, because the hard links to directories work differently compared to hard links to files, i.e., they cannot be created explicitly like the `export` method in the `Node` class does, but are automatically created in subdirectories. The first step in the export method is to delete anything that is stored at the export path of the directory node using the `remove` function. Contrary to the `export` method of the `Node` class, this call does not delete a directory at the export path. The reason for this exception is explained in Section 5.4.6.2. In general, this call does nothing because no file or directory exists at this path for empty images that are the default target for exports.

The remainder of the overwritten method is actually pretty straightforward: it creates an empty directory using the `mkdir` function unless it already exists (Lines 723 and 724), calls the `export` method of all its child nodes (Lines 726 to 727), and finally applies the metadata to the directory. It is important that the metadata is applied after all child nodes are exported, because the export of child nodes can modify the metadata of the exported directory.

The `OpaqueDirectory` class again overwrites the `export` method, but it delegates the actual exporting to the overwritten method of its superclass `Directory`. It calls a hook method in the `Exporter` instance before and after the directory is exported.

```
548 class Directory(Node):
        ...
719     def export(self, datastore, exporter):
720         path = exporter.getPath(self)
721 
722         fileops.remove(path, keep_directories=True)
723         if not os.path.exists(path):
724             os.mkdir(path)
725 
726         for child in self._children:
727             child.export(datastore, exporter)
728 
729         self.stats.export(path)
 ⋮
815 class OpaqueDirectory(Directory):
        ...
828     def export(self, datastore, exporter):
829         exporter.handleOpaqueDirectoryPre(self)
830         Directory.export(self, datastore, exporter)
831         exporter.handleOpaqueDirectoryPost(self)
 ⋮
839 class File(Node):
        ...
850     def do_export(self, datastore, exporter):
851         path = exporter.getPath(self)
852         datastore.copyData(exporter, self)
853         self.stats.export(path)
854         return path
 ⋮
860 class Whiteout(Node):
        ...
890     def do_export(self, datastore, exporter):
891         return exporter.handleWhiteout(self)
```

Listing 5.7 Virtual Machine Export – Part 2. This listing shows the specific export methods for node types not shown in Listing 5.6: directories, opaque directories, files, and whiteout files.

This enables the `Exporter` instance to export the opaque directory correctly. An exemplary hook method is shown in Listing 5.10.

The `do_export` method of the `File` class is again straightforward. It uses the `copy` method of the Data Store (Line 92 in Listing 5.3) to copy the file's contents to the path determined by the node's `Exporter` instance (Lines 851 to 852). Afterwards, the metadata is applied using the `export` methods of the node's `Stats` instance and the path of the exported node is returned.

Whiteout files are the final node type: they are exported by a call to the `handle-Whiteout` method of the node's `Exporter` instance. Its implementation is shown in Listing 5.9.

Listing 5.8 shows the implementation of the `remove` function that will check whether a file, special file, i.e., a symbolic link, device file, or named pipe, or a directory exists at a path passed as argument and deletes it in this case. Using the optional keyword argument `keep_directories` the function can be instructed to remove only files and special files, but keep directories.

Files and special files can be removed with the `remove` function of Python's `os` module (Line 60), whereas directories are removed using the `rmdir` function. Unfortunately,

```python
55 def remove(path, keep_directories=False):
56     if path.endswith('/'):
57         path = path[:-1]
58     if os.path.lexists(path):
59         if os.path.islink(path) or not os.path.isdir(path):
60             os.remove(path)
61         else:
62             if not keep_directories:
63                 shutil.rmtree(path)
```

Listing 5.8 Removing Files and Directories. This listing shows a helper function that checks whether a file, special file, or directory at a given path exists and deletes it in this case.

the latter only works for empty directories. The `rmtree` function in the `shutil` module can be used to recursively delete entire directory trees (Line 63). However, this works only for directories, not for symbolic links to directories. The remainder of the function deals with some of the peculiarities of Python's standard library. In Lines 56 to 57 a trailing slash is removed, because the `lexists` function used to check if a file or directory with a given name exists. Even if a file exists, it returns `False` if the file name is passed with a trailing slash. This happens a file exists at the export path of a directory node, because the node will return its path with a trailing slash – it is a directory after all. The next pitfall is the difference between the `exists` and `lexists` functions: the former follows links and thus returns `False` for an existing symbolic link with a non-existing target, whereas the latter does not follow the link and returns `True`. Finally, the `isdir` function follows links as well and thus returns `True` for symbolic links to directories. This test must thus be accompanied by a call to `islink` (Line 59).

```python
class Exporter(object):
    def __init__(self, target):
        self._target = os.path.abspath(target)
        self.linkcache = dict()

    def getPath(self, node):
        return self._target + '/' + node.path

    def handleWhiteout(self, node):
        pass

    def handleOpaqueDirectoryPre(self, node):
        pass

    def handleOpaqueDirectoryPost(self, node):
        pass
```

Listing 5.9 Virtual Machine Export – Default Exporter. This listing shows the default exporter regular virtual machine images: the `Exporter` class.

As already written above, the `Exporter` instance (see Listing 5.9) is used to calculate absolute paths for exporting nodes, to store the list of already exported nodes for hard link detection and to export whiteout files and opaque directories for specific union mount implementations. The type of exporter used influences the result of the export process, i.e., by controlling how whiteout files and opaque directories are exported and by computing the path for each exported node. The target path of the output process is passed as argument to the exporter's constructor and stored in the instance for use in the `getPath` function. Additionally, the constructor initializes the dictionary of already exported files (Lines 9 to 10).

The `getPath` method simply concatenates the target path with the node's path returned by the property of the same name. The default implementation `Exporter` exports manifest as regular virtual machine images. As such, the exported images should contain neither whiteout files nor opaque directories. The helper methods used to create the special marker files are thus empty (Lines 12 to 19).

In order to export a manifest for use as layer in a composite disk image, whiteout files and opaque directories are import for the correct composition. The `AUFSExporter` class shown in Listing 5.10 is able to export the required marker files for the Image Compositor implemented using aufs. If another union mount implementation is used for image composition, a corresponding export must be written for the Image Store.

An extended version of the `getPath` method is required to handle whiteout files correctly. As written above, the Image Store uses generic whiteout nodes to represent whiteout files independently of the union mount implementation used. The generic whiteout nodes' name does therefore not include the prefix ".wh." that is used by aufs. In Lines 32 to 33 the prefix is added if the passed node is a whiteout node.

Whiteout files are created by the `handleWhiteout` method. The method creates an empty file by opening the file in append mode and closing it immediately afterwards (Line 40). Afterwards it applies the metadata and returns the path of the exported whiteout file. Because `Whiteout` instances use the inherited `export` method, the hard link detection is also applied to whiteout files. Consequently, all whiteout files are just hard links to a single empty file – exactly the way aufs creates whiteout files.

Opaque directories are regular directories that contain a special marker file in aufs. After the directory is exported, the post export hook `handleOpaqueDirectoryPost` is called by the `export` method of the `Whiteout` class to create the marker file named "`.wh..wh..opq`". The markers generated by aufs are hard links to the same single empty file that is used as whiteout file. This file is named "`.wh..wh.aufs`" and located in the root directory of the manifest. For better comprehensibility, it is called *whaufs* in the remainder of this section. Unfortunately, the default mechanism to create hard links does not work for the opaque directory marker, because `OpaqueDirectory` overwrites `export`. Additionally, it is not possible to blindly create a hard link to *whaufs* , because the order in which the files are exported is not guaranteed. Even if *whaufs* is not exported yet, another whiteout file might have already been exported. This cannot be checked without the `st_ino` value of the marker file in the original image, which is not part of the manifest because the marker file is not represented as a node of its own.

The `handleOpaqueDirectoryPost` method first checks whether *whaufs* is already exported (Line 49). If it is found, it simple creates a link to it (Line 65). Otherwise, the method loops over the children of the manifest's root node until it finds the node

```
28  class AUFSExporter(Exporter):
29      def getPath(self, node):
30          path = Exporter.getPath(self, node)
31          if isinstance(node, Whiteout):
32              splitted = path.rsplit('/', 1)
33              return splitted[0] + '/.wh.' + splitted[1]
34          else:
35              return path
36
37      def handleWhiteout(self, node):
38          path = self.getPath(node)
39          fileops.remove(path)
40          open(path, 'a').close()
41          node.stats.export(path)
42          return path
43
44      def handleOpaqueDirectoryPost(self, node):
45          whaufs_path = self._target + '/.wh..wh.aufs'
46          dir_path = self.getPath(node)
47          marker_path = dir_path + '/.wh..wh..opq'
48          fileops.remove(marker_path)
49          if not os.path.exists(whaufs_path):
50              whaufs_node = None
51              for child in node.root()._children:
52                  if isinstance(child, Whiteout) and child.name == '.wh.aufs':
53                      whaufs_node = child
54                      break
55              if whaufs_node:
56                  if (whaufs_node.stats.st_ino in self.linkcache):
57                      os.link(self.linkcache[whaufs_node.stats.st_ino],      ↩
        marker_path)
58                  else:
59                      open(marker_path, 'a').close()
60                      whaufs_node.stats.export(marker_path)
61                      self.linkcache[whaufs_node.stats.st_ino] = marker_path
62              else:
63                  print 'failed to lookup whaufs node'
64          else:
65              os.link(whaufs_path, marker_path)
66          node.stats.export(dir_path)
```

Listing 5.10 Virtual Machine Export – aufs Exporter. This listing shows the exporter for layered virtual machine images using aufs: the `AUFSExporter` class.

representing *whaufs* (Lines 50 to 54). Note that this is a whiteout node, so its name in the manifest is actually "`.wh.aufs`" without the whiteout file prefix. After it has found the node, the method checks whether any whiteout file has already been exported by searching for an entry with *whaufs'* `st_ino` value in the dictionary of exported files (Line 56). In an entry is found, a link to the path stored in this entry is created (Line 57). Otherwise, an empty file is created as opaque directory marker, the metadata of *whaufs* is applied to the marker file, and the path of the marker file is registered with *whaufs'* `st_ino` value in the dictionary of exported files (Lines 59 to 61), so it can be found by the `export` method in the `Node` class when a whiteout file needs to be exported.

Irrespective of the way the opaque directory was created, in all three cases a file is added to the directory and thus its time of the last modification is updated automatically. Therefore, the metadata of the directory node needs to be applied to the directory again (Line 66).

5.4.3.3 On the Fly Modifications during Export

In some cases it is useful to be able to make minor modifications during the export of a virtual machine. Generally, such modifications are made by creating a clone of the virtual machine and updating the clone using the normal means provided by the Image Store. Using the update procedures presented in Section 5.4.6.1, updates can be done in an efficient manner. However, there are some modifications that are required only for a specific instance of a virtual machine during its execution. Examples of such modifications are the installation of SSH public keys to allow a user to log in remotely and configurations changes to adapt a virtual machine to an execution environment, or to configure installed software for active use.

The Image Store provides a very efficient way to make such minor changes to a virtual machine without cloning and updating a virtual machine: *Patch* nodes. This is a special type of node that is added to a manifest to replace the content of a file during export. A Patch contains not only metadata like the other node types, but also the modified content of the file it replaces. Note that contrary to the general idea of the Image Store, a manifest including Patch nodes actually contains data. The modifications introduced to the manifest by Patch nodes are volatile, i.e., they affect the current export process, but are not persisted in anywhere (except in the exported image). The actual manifest in the Metadata Store is never changed by the use of Patch nodes.

The generation of Patch nodes takes place before the actual export process starts. The server component `missrv` executes a set of scripts when the remote client component `misrcl` requests a manifest. These scripts are configured when the server is started. Each script can generate new content for a single file based the on the old content of the file and a set of arguments that are passed from the client. If the script returns new content, a Patch node with the generated content is added to the manifest in place of the original node. Otherwise, the original node is left untouched. During export, the generated content stored in the Patch node is copied to the target virtual machine image instead of the original content.

Consider a script that generates a custom `authorized_keys` file for SSH, i.e., a file containing the public keys of all users that are allowed to remotely log in as the particular user in whose `.ssh` directory the file resides. That script loops over all parameters passed by the client and checks if a specific parameter, e.g., `SSH_PUBKEY`, exists. If that parameter exists, it returns its value to the server component that will create a Patch node with the returned value as its content. Thus, the value of the parameter provided by the client is exported into the `authorized_keys` file in the exported image. On the other hand, if the parameter does not exist, the script returns an error code to the server component and the actual node representing the `authorized_keys` file in the manifest is not replaced.

5.4.4 Manifest Operations

Operations on manifests are an import part of the Image Store. They are the basis of the advanced operations like efficient updates or content filtering that are described in detail in Section 5.4.6. This section contains a description of the manifest operations provided by the Image Store. The description starts with basic filter operations that can be used to select or reject nodes based on specific conditions. Afterwards, the more complex *Merge* and *Diff* operations are described.

5.4.4.1 The Filter Operations

The filter operations enable removing nodes from a manifest using on or more different conditions. The filter operations come in two flavors: *Select* and *Reject*[14]. The Select operation removes all nodes from the manifest that do not satisfy the conditions, whereas the Reject operation removes all nodes from the manifest that do satisfy the conditions. Listing 5.11 shows the implementation of these filter operations.

The Select and Reject operations are implemented by methods of the same name in the `Directory` class (Lines 796 to 800) that do nothing more than delegating the filter operation to the `_select_reject` method, which implements both operations. Both methods require a condition as argument and optionally accept a Boolean value to control whether the filter operations work in *recurse-first* mode (`True`) or in the default *filter-first* mode (`False`). Both arguments are passed on to `_select_reject` together with an additional Boolean value that controls whether the filter works in select or reject mode. Conditions are functions that accept a node as only argument and return `True` if the node satisfies the condition and `False` otherwise. An easy way to create such conditions is described below.

The `_select_reject` method recursively calls itself for all child nodes of the directory node type using the inner function `recurse`. Depending on whether the depth-first mode or the filter-first mode is selected, the recursion is performed before (Line 785) or after (Line 794) the filtering. The actual filtering of the child nodes is done in Lines 787 to 790 using two list comprehensions that evaluate the condition for the each child. Note that this method works destructively, i.e., it modifies the original manifest in memory instead of creating a copy of the selected nodes for performance reasons. If this is not the desired behavior, the manifest has to be clone manually before the filter operation is executed.

Removing child nodes from a directory node can cause a discrepancy between the directory node and its metadata: the `st_nlink` field in its `Stats` instance that stores the number of hard links to the directory. In Linux it is not possible to manually create additional hard links to directories, as this could create cycles in the directory hierarchy of the file system. One might expect that the number of hard links of a directory is a fixed value for this reason. However, the file system creates hard links to directories every time a subdirectory is created: every subdirectory contains an

[14] The filter operations are inspired by the two methods with the same name in the *Enumerable* class of the core library of the *Ruby* programming language.

```python
class Manifest(object):
    ...
    def _remove_empty_directories(self):
        self.root.reject(
            lambda node: isinstance(node, Directory) and len(node._children) == 0,
            True)

    def reject(self, condition):
        self.root.reject(condition)
        self._remove_empty_directories()

    def select(self, condition):
        self.root.select(selector.cor(condition,
            lambda node: isinstance(node, Directory)))
        self._remove_empty_directories()
⋮
class Directory(Node):
    ...
    def _fix_st_nlink(self):
        self.stats.st_nlink = 2 + len(
            [child for child in self._children if isinstance(child, Directory)])

    def _select_reject(self, condition, select=True, depth_first=False):
        def recurse():
            for child in self._children:
                if isinstance(child, Directory):
                    child._select_reject(condition, select, depth_first)

        if depth_first:
            recurse()

        if select:
            self._children = [child for child in self._children if condition(child)]
        else:
            self._children = [child for child in self._children if not condition(child)]
        self._fix_st_nlink()

        if not depth_first:
            recurse()

    def reject(self, condition, depth_first=False):
        self._select_reject(condition, False, depth_first)

    def select(self, condition, depth_first=False):
        self._select_reject(condition, True, depth_first)
```

Listing 5.11 The Select and Reject Filter Operations. This listing shows the methods that implement the Select and Reject filter operations for manifests.

"`..`" directory entry that points to the parent directory. This is actually a hard link to the parent directory and thus increases its `st_nlink` value. Besides the hard links in subdirectories, there are two more hard links for each directory: the directory entry of the directory itself in the parent directory and the "`.`" directory entry in the directory itself. At every point in time, a directory thus has a `st_nlink` value equal to the number of subdirectories plus two. The `_fix_st_nlink` method of the `Directory` class called after filtering updates the `st_nlink` value according to this rule.

In the `Manifest` classes wrappers for the `select` and `reject` methods exist that call the corresponding method on the root node of the manifest. The `reject` method passes the condition unmodified, while the `select` method passes a compound condition that combines the original condition with another condition that is satisfied for any directory using a logical *or* operator. The reason for this specific compound condition will become apparent when a few example conditions are described below.

After the filtering operation is finished both methods remove empty directories using an internal method. The removal of those empty directories is itself implemented as an additional Reject operation that discards all directory nodes without child nodes (Line 128). Note that this operation has to be processed in the recurse-first mode, otherwise it will only remove a single level of empty leaf directory nodes. Figure 5.13a shows an exemplary tree of nodes consisting of three directory nodes to clarify the problem. If the operation is processed in filter-first mode, the child notes of `DirectoryA` will be filtered first. Because `DirectoryB` is not empty at that point, it will not be removed. Afterwards, the child nodes of Directory B will be filtered. This removes `DirectoryC`, because it is empty. The filter operation is finished at that point and the empty `DirectoryB` is still part of the tree (see Figure 5.13b).

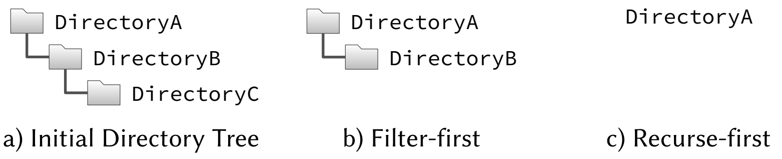


Figure 5.13 Exemplary Directory Hierarchy. An exemplary directory hierarchy to used to demonstrate the differences between the recurse-first and filter-first modes of the Reject operation for removing empty directories.

In recurse-first mode, the child nodes of `DirectoryC` are filtered first. Afterwards, the child nodes of `DirectoryB` will be filtered. This removes `DirectoryC` just like in the filter-first mode, because it is empty. Finally, the child nodes of `DirectoryA` are filtered. This removes `DirectoryB`, because it is empty at this point. The resulting tree contains only `DirectoryA` (see Figure 5.13c), which is correct although `DirectoryA` is empty in this example, because it is the root node of the tree.

Listing 5.12 shows a set of helper functions that ease the creation of filters. The `condition` function is used to convert file names with wildcard characters into conditions that can be used with the filter operations. The file names are converted into a valid regular expression and compiled using the helper method `to_regex` (Line 8). A lambda function that tries to match the path of a passed node with the compiled regular expression is then returned to the caller (Line 9).

In addition to the `condition` function, there are three compound conditions `cor`, `cand`, and `cnot` that combine multiple conditions using the logical *or*, *and*, and *not* operation, respectively. Both `cor` and `cand` convert a list of conditions using the `condition` function in a list comprehension. This is the reason for simply returning

```
 4  def condition(cond):
 5      if type(cond) == types.FunctionType:
 6          return cond
 7      elif type(cond) == str:
 8          regex = to_regex(cond)
 9          return lambda node: regex.search(node.path) is not None
10
11  def cor(*conditions):
12      conditions = [condition(cond) for cond in conditions]
13      def test(node):
14          for condition in conditions:
15              if condition(node): return True
16          return False
17      return test
18
19  def cand(*conditions):
20      conditions = [condition(cond) for cond in conditions]
21      def test(node):
22          retval = True
23          for condition in conditions:
24              if not condition(node):
25                  retval = False
26          return retval
27      return test
28
29  def cnot(condition):
30      return lambda node: not condition(node)
31
32  def to_regex(string):
33      string = string.strip()
34      regex_src = string.replace('.', '\.').replace('?',
        '[^/]').replace('**', '.+') \
35          .replace('*', '[^/]+')
36      regex_src = regex_src + '$'
37      if string[0] == '/':
38          regex_src = '^' + regex_src
39      return re.compile(regex_src)
```

Listing 5.12 Helper Functions for Definition Conditions. This listing defines a set of helper functions that ease the creation of conditions for the manifest filter operations.

conditions of the `function` type directly to the caller in the `condition` method (Lines 5 to 6). Both functions return a test function that implements the logical *or* and *and* operations, respectively. On the other hand, the `cnot` function accepts only a single condition[15] and returns a lambda function that returns the logically negated result of the condition function.

The `to_regex` function is responsible to convert a string with wildcard characters into a valid regular expression and return a compiled regular expression object. It iteratively replaces wildcard characters with a corresponding regular expression snippet using a chain of `replace` invocations. Additionally, it masks the "`.`" character that has a special meaning in regular expressions (Lines 34 to 35). The regular expression is

[15] That can of course in turn be another compound condition.

unconditionally *anchored* at the end of the string (Line 36). This is a reasonable choice, because the file name is probably the most prevalent search criterion. If this behavior is not desired, the "`**`" wildcard can be appended to the string to render this anchor useless. If the string is an absolute path, i.e., it is starting with a "`/`" character, the regular expression is additionally anchored at the start of the string (Line 38). The list of supported wildcard characters, their replacement, and their meaning are shown in Table 5.10.

Table 5.10 Supported Wildcard Characters. This table lists the supported wildcard characters (Char), their corresponding regular expression snippet (RegEx), and their meaning.

Char	RegEx	Meaning
`?`	`[^/]`	A single arbitrary character excluding the path separator.
`*`	`[^/]+`	Any number of arbitrary characters excluding the path separator. This wildcard only matches a single component of the part.
`**`	`.+`	Any number of arbitrary characters including the path separator. This wildcard can be used to match multiple components of the path at once.

A few examples of wildcard use are given in Section 5.4.6.3. Note that this is just a proof of concept implementation with a limited set of wildcard characters. Besides the limited functionality, the implementation is probably not very efficient, because even static strings are matched using regular expressions, although string comparisons are faster. Furthermore, for a large set of conditions it is perhaps inefficient to use distinct regular expressions. Intelligent grouping of conditions based on shared prefixes can possibly improve the performance of the filtering solution by reducing the amount of conditions that has to be checked for every node.

Listing 5.13 shows 5 exemplary filter operations with different conditions defined using the helper functions described above. The first Select operation selects all file nodes that can be identified as python scripts based on their file name. Irrespective of whether the depth-first mode is used or not, if this condition is passed unmodified to the `select` function of the root directory node, it will only return direct children of the root directory node that satisfy the condition, i.e., python scripts in the root directory. None of the directory child nodes of the root directory node will satisfy the condition and thus they are not selected, although they might in turn have child nodes that satisfy the condition. The composite condition built in the `select` method of the `Manifest` class (Lines 136 to 137 in Listing 5.11) additionally selects all directories, so all file nodes in the entire manifest that satisfy the original condition are returned. This is also the main reason for filtering empty directories, because it the compound condition used in the Select operation selects every directory node in the manifest even if it does not contain a single file matching the original condition.

It is important to note that a Select operation not only return a list of nodes, but an entire tree of nodes. This is necessary, because the selected nodes only know their direct parent, but not their position in the tree of nodes. By returning only the selected nodes the entire directory structure would be lost. Additionally, the resulting tree of

```python
manifest.select(condition('*.py'))
manifest.select(lambda node: isinstance(node, SymbolicLink))
manifest.select(cand('*.py', lambda node: node.stats.st_size >= 4096))
manifest.reject(cor('id_rsa', '.bash_history'))
manifest.reject(cand(lambda node: isinstance(node, File), cor('id_rsa',
    '.bash_history')))
```

Listing 5.13 Manifest Filter Operation Examples. A few examples of the conditions that can be used with the manifest filter operations.

node is a valid manifest and can thus be combined with other manifests using the Merge and Diff operations.

The second Select operation in Listing 5.13 selects all symbolic link types using a lambda function. Because this is already a valid condition, the `condition` helper function does not need to be used, although passing the lambda function to the `condition` function would do no harm. The third Select operation uses a compound condition that implements a logical *and* operator. It selects all file nodes that can be identified as python scripts based on their file name and additionally have a size of more than 4,096 bytes. This time, neither the file name pattern nor the lambda function needs to be passed to the `condition` function explicitly, because the `cand` function takes care of this. Note that both the `cand` and `cor` functions support and arbitrary number of conditions as arguments.

The fourth filter operation is a Reject operation. It is an actual example of how sensitive content can be filtered from a virtual machine image[16]. This Reject operation removes nodes of any type that are named either "`id_rsa`" or "`.bash_history`", i.e., an SSH private key and a history file of the bash shell, using a compound condition that implements a logical *or* operator. The last filter operation combines the condition of the fourth operation with a node type check using a `cand` compound condition. Consequently, this filter operation removes only file nodes named either "`id_rsa`" or "`.bash_history`" instead of nodes of any type like the fourth filter operation.

5.4.4.2 The Merge Operation

The Merge operation combines two manifests in exactly the same way as a union mount does with two directories. The result of merging a manifest M_U (the *upper* manifest) onto a manifest M_L (the *lower* manifest) is the manifest M_M (the merged manifest). It is computed by applying the following rules on every pair of corresponding nodes from M_U and M_L with the same path according:

1. A directory node in M_U

    a) Is merged with its corresponding directory node in M_L by recursively applying this rules on their child nodes. This rule also applies to an opaque directory node in M_L. For an opaque directory node in M_U see Rule 4.

[16] More information about content filtering is given in Section 5.4.6.3

   b) Is included in M_M unless its corresponding node in M_L is a directory or opaque directory.

2. A file node (regular file, symbolic link, device file, or named pipe) in M_U is included in M_M irrespective of the type of its corresponding node in M_L.
3. A whiteout node in M_U is included in M_M irrespective of the type of its corresponding node in M_L.
4. An opaque directory node in M_U is included in M_M irrespective of the type of its corresponding node in M_L. This rule take precedence over Rule 1 for an opaque directory node in M_U.
5. Any node in M_U without a corresponding node M_L is included in M_M.
6. Any node in M_L without a corresponding node M_U is included in M_M.

It is important to note that this operation is not commutative, i.e., merging manifest A onto manifest B results in a different manifest compared to merging manifest B onto manifest A.

Listing 5.14 shows the implementation of the Merge operation in the Image Store. The `merge_onto` method merges the node it is called on (`upper`) onto the node passed in as the `lower` argument. Note that these methods make use of the possibility to choose a custom name for the reference to the *current instance* in Python that is called `self` just by convention. It is named `upper` in the `merge_onto` methods to clarify it is referencing nodes from the upper manifest. The Merge operation is initiated by calling the `merge_onto` method of the upper manifest's root directory node and passing the lower manifest's root directory node as the `lower` argument. A wrapper method exists in the `Manifest` class exists that takes care of this (not shown).

The most important part of the Merge operation is the `merge_onto` method in the `Directory` class that implements Rules 1, 5, and 6. Rule 1b is implemented in Lines 598 to 599 by returning a *deep copy* of the upper directory node that replaces the lower node. A deep copy is a copy of a directory node that includes all child nodes (recursively), whereas a *shallow copy* of a directory node includes none of the child nodes, but only the directory node itself. The implementation of these copy operations is shown in Listing 5.15.

In Line 601 a shallow copy of the upper directory node is created that will be the resulting node of this call to `merge_onto`. At this point, it contains only the metadata of the upper directory node, but no children. For easier lookup of child nodes two dictionaries are created from both directory nodes' lists of children (Lines 603 to 604) and a set of child names is created by computing the union of the names (keys) in those dictionaries (Line 606). A for loop is used to iterate over all children names in the set, retrieve the associated child nodes (Lines 609 to 610) and apply the appropriate rule: if child nodes with the current name exist in both directory nodes, Rule 1a is applied and the two child nodes are merged by recursively calling the `merge_onto` method of the upper child node and passing the lower child node as argument (Line 613). Contrary, if only a single child node with the current name exists either in the upper or lower directory node, Rule 5 or 6 is applied by creating a deep copy of the child

```
231  class Node(object):
         ...
287      def merge_onto(upper, lower):
288          return copy(upper)
 ⋮
548  class Directory(Node):
         ...
597      def merge_onto(upper, lower):
598          if not isinstance(lower, Directory):
599              return deepcopy(upper)
600
601          retval = copy(upper)
602
603          u_children = upper.children_as_dict
604          l_children = lower.children_as_dict
605
606          names = set(u_children.keys() + l_children.keys())
607
608          for name in names:
609              u_child = u_children.get(name, None)
610              l_child = l_children.get(name, None)
611
612              if (u_child and l_child):
613                  child = u_child.merge_onto(l_child)
614              elif (u_child):
615                  child = deepcopy(u_child)
616              else:
617                  child = deepcopy(l_child)
618
619              child.add_to(retval)
620
621          retval._fix_st_nlink()
622          return retval
 ⋮
815  class OpaqueDirectory(Directory):
         ...
826      def merge_onto(upper, lower):
827          return deepcopy(upper)
```

Listing 5.14 The Merge Operation. This listing shows the methods that implement the Merge operation for manifests.

node (Lines 615 and 617, respectively). A deep copy has to be created instead of a shallow copy because the child node might be a directory node itself. The merged or copied child node is then added to the resulting directory node (Line 619). After the loop is finished, the `st_nlink` value of the resulting directory node is fixed, because it likely does not correspond to the number of its child directory nodes any more, and the node is returned to the caller.

The Rules 2 and 3 (Lines 287 to 288) as well as 4 (Lines 826 to 827) are easy to implement: in all three cases a copy of the upper node needs to be returned as result of the merging. Nevertheless, there is an important difference: the `merge_onto` implementation in `OpaqueDirectory` needs to return a deep copy of upper node, whereas for the implementation in `Node` a shallow copy is sufficient.

The Merge operation (as well as the Diff operation described below) relies on a mechanism to create both deep and shallow copies of nodes. Python provides a framework for creating such copies that relies on the implementation of `__copy__` and `__deepcopy__` methods as shown in Listing 5.15.

```
143 class Stats(object):
        ...
161     def __copy__(self):
162         return Stats(self)
163
164     def __deepcopy__(self):
165         return self.__copy__()
 ⋮
231 class Node(object):
        ...
358     def __copy__(self):
359         retval = self.__class__(self.name)
360         for key in self.__slots__:
361             if (key[0] == '_'):
362                 continue
363
364             if hasattr(self, key):
365                 setattr(retval, key, copy(getattr(self,key)))
366
367         return retval
368
369     def __deepcopy__(self, memo):
370         return self.__copy__()
 ⋮
548 class Directory(Node):
        ...
628     def __deepcopy__(self, memo):
629         retval = self.__copy__()
630         memo[id(self)] = retval
631         for child in self._children:
632             deepcopy(child, memo).add_to(retval)
633         return retval
```

Listing 5.15 Deep and Shallow Node Copies. This listing shows the implementation of deep and shallow copies of nodes and their metadata.

For both the `Stats` and the `Node` class both copy operations are identical, because none of the fields copied contains objects that would require a distinction between a shallow and a deep copy. Thus, the `__deepcopy__` method just delegates the copying to the `__copy__` method. A `Stats` instance can be copied by creating a new instance and passing the existing one to the constructor of the copy. This works because Python uses *duck typing* and the `Stats` instance has the all the fields that the constructor tries to copy from the passed object.

The implementation of `__copy__` in the `Node` class is a bit more complex. As a first step, a new instance of the correct subclass of `Node` is created (Line 359). The method then iterates of all instance variables listed in the `__slots__` class variable and copies the values to the new instance (Lines 366 to 367) if they are not `None`. However, there is an exception: all fields having a name that starts with an underscore letter are not copied. This is used to prevent copying the `_parent` field to the new instance, because the parent node of the instance is not also the parent node of the new instance. Instead, the correct parent node is set when the new instance is added to the tree of nodes using the `add_to` method.

This exception also allows subclasses of `Node` to define complex fields containing references to other objects. Such fields make a distinction between a shallow and a deep copy necessary. By skipping these fields, a subclass can provide appropriate methods for creating shallow and deep copies that can still use the inherited `__copy__` method to copy the inherited fields. The `_children` field of the `Directory` class is an example of such a field. For `Directory` instances, the `__copy__` method copies just the directory node itself and the `__deepcopy__` method also copies the child nodes. The former does not need to be implemented, because the inherited method is sufficient. It skips the list of children when it initializes the new instance because of its name. The constructor of `Directory` ensures that it is still initialized with a valid value – in this case an empty list. The `__deepcopy__` method first does a shallow copy of the directory (Line 629). Afterwards, it creates a deep copy of each child node and adds the copy of the child node to the copy of the directory using a for loop (Lines 631 to 632)[17].

5.4.4.3 The Diff Operation

The Diff operation computes the differences between two manifests and generates a new manifest that describes these differences. The differences between a manifest M_C (the *current* manifest) and a manifest M_B (the *baseline* manifest) is the manifest M_D. It is computed by applying the following rules on every pair of corresponding nodes from M_C and M_B with the same path. In this set of rules opaque directory nodes are treated as directory nodes.

1. A directory node in M_C
   a) Is distinguished from its corresponding directory node in M_B by recursively applying these rules to their child nodes. The directory node in M_C is included in M_D if either its metadata differs from the metadata of its corresponding node in M_B or any of its child nodes[18] is included in M_D.
   b) Is included in M_D if its corresponding node in M_B is not a directory node.
2. A file node (regular file, symbolic link, device file, or named pipe) in M_C is included in M_D if it is not equal to its corresponding node in M_B. Equal here means equal type, metadata, and content (if any).
3. A whiteout node in M_C is included in M_D if its corresponding node in M_B is not a whiteout node.
4. Any node in M_C without a corresponding node in M_B is included in M_D.
5. For any node in M_B without a corresponding node in M_C a whiteout node with the same path is included in M_D.

[17] The `memo` variable is used to prevent recursive loops while copying complex object graphs. Such loops can occur if recursive objects are copied, i.e., objects that reference itself directly or indirectly.

[18] This includes whiteout nodes created for any of the corresponding directory node's child nodes due to Rule 3.

The manifest M_D generated by applying the Diff operation on the manifests M_C and M_B is basically a blueprint of how to modify M_B to become equal to M_C. Consequently, the result M_M of merging M_D onto M_B using the Merge operation is equal to M_C. It is important to note that the Diff operation is not commutative, i.e., computing the difference between manifest A and manifest B results in a different manifest compared to computing the difference between manifest B and manifest A.

Listing 5.16 shows the implementation of the Diff operation in the Image Store. The `diff_to` method computes the difference between the node it is called on (`current`) and the node passed in as the `baseline` argument. Note that these methods (like the `merge_onto` methods implementing the Merge operation) use a custom name instead of `self` for referencing the instance they are called on. The reference is named `current` in the `diff_to` methods to clarify it is referencing nodes from the current manifest. The Diff operation is initiated by calling the `diff_to` method of the current manifest's root directory node and passing the baseline manifest's root directory node as the `baseline` argument. A wrapper method exists in the `Manifest` class exists that takes care of this (not shown).

The most important part of the Diff operation is the `diff_to` method in the `Directory` class that implements Rules 1, 4, and 5. Rule 1b is implemented in Lines 682 to 683 by returning a *deep copy* of the current directory node that is added to the resulting manifest by the caller. In Line 685 a shallow copy of the current directory node is created that will be the resulting node of this call to `diff_to`. At this point, it contains only the metadata of the current directory node, but no children. For easier lookup of child nodes two dictionaries are created from both directory nodes' lists of children (Lines 687 to 688) and a set of child names is created by computing the union of the names (keys) in those dictionaries (Line 690).

A for loop is used to iterate over all children names in the set, retrieve the associated child nodes (Lines 693 to 694) and apply the appropriate rule: if corresponding child nodes exist in both directory nodes, Rule 1a is applied and the difference between the two child nodes is computed by recursively calling the `diff_to` method of the current child node and passing the baseline child node as argument (Line 697). Contrary, if only a single child node exists either in the current or baseline directory node, Rule 4 or 5 is applied depending on which directory node contains the child node by either creating a deep copy of the current child node (Line 699) or creating a corresponding whiteout node (Line 701), respectively. A deep copy has to be created instead of a shallow copy in the former case because the current child node might be a directory node itself. The content of the `node` variable, which is either the result of recursively calling `diff_to`, a copy of the current node or a whiteout node, is then added to the resulting directory node (Line 704) unless it is `None`. The `node` variable can only be `None` as a result of calling `diff_to` for two equal child nodes.

After the loop is finished, the number of child nodes in the resulting directory node is checked. If it is zero, all child nodes are equal. The resulting manifest of a Diff operation must not contain empty directory nodes unless the metadata of the directory node is different from baseline in current. This is ensured by the second condition in Line 706 that causes `None` to be returned if the metadata of both directory nodes is equal. Comparing the `Stats` instances is sufficient here, because the nodes' types

```
231 class Node(object):
        ...
296     def diff_to(current, baseline):
297         if (current == baseline):
298             return None
299         else:
300             return copy(current)
     ⋮
548 class Directory(Node):
        ...
681     def diff_to(current, baseline):
682         if not isinstance(baseline, Directory):
683             return deepcopy(current)
684
685         retval = copy(current)
686
687         c_children = current.children_as_dict
688         b_children = baseline.children_as_dict
689
690         names = set(c_children.keys() + b_children.keys())
691
692         for name in names:
693             c_child = c_children.get(name, None)
694             b_child = b_children.get(name, None)
695
696             if (c_child and b_child):
697                 child = c_child.diff_to(b_child)
698             elif c_child:
699                 child = deepcopy(c_child)
700             else:
701                 child = Whiteout(name)
702
703             if child:
704                 child.add_to(retval)
705
706         if len(retval._children) == 0 and current.stats == baseline.stats:
707             return None
708
709         retval._fix_st_nlink()
710         return retval
```

Listing 5.16 The Diff Operation. This listing shows the methods that implement the Diff operation for manifests.

and names are equal[19]. Finally, the `st_nlink` value of the resulting directory node is fixed, because it very likely does not correspond to the number of its child directory nodes any more, and the node is returned to the caller.

The Rules 2 and 3 are implemented in the `diff_to` method of the `Node` class. It compares the current node with the baseline node and returns `None` if the nodes are equal or a copy of the current node otherwise. A shallow copy is sufficient in this case.

The implementation of the Diff operation compares needs to compare nodes (Line 297) and instances of the `Stats` class storing nodes' metadata (Line 706). This requires

[19] Both nodes must be directory nodes, otherwise the method would return in Line 683, and their names must be identical, because `diff_to` is only called for nodes with the same name (Line 697.

specific comparison methods for the affected classes is invoked by Python whenever an instance of the Node class (or its subclasses) or the Stats classes is compared using either the == or != operator. In the former case the __eq__ method is invoked, whereas in the latter case the __ne__ is invoked. These methods are shown in Listing 5.17.

```
143  class Stats(object):
         ...
167      _eq_slots = ['st_mode', 'st_nlink', 'st_uid', 'st_gid', 'st_size',
168          'st_atime', 'st_mtime']
169
170      def __eq__(self, stats):
171          for slot in self._eq_slots:
172              if not getattr(self, slot, None) == getattr(stats, slot, None):
173                  return False
174          return True
175
176      def __ne__(self, stats):
177          return not self == stats
 ⋮
231  class Node(object):
         ...
338      def __eq__(self, node):
339          if self.__class__ != node.__class__:
340              return False
341
342          for key in self.__slots__:
343              if (key[0] == '_'):
344                  continue
345
346              if not getattr(self, key, None) == getattr(node, key, None):
347                  return False
348
349          return True
350
351      def __ne__(self, node):
352          return not self == node
```

Listing 5.17 Comparing Nodes. This listing shows the methods that implement the comparison of nodes required for the implementation of the Diff operation.

The __eq__ method of the Stats class iterates over a subset of its fields using the list in _eq_slots and compares the values of each field in both self and stats, i.e., the instance the method is called on and the second instance it should be compared with (Lines 171 to 172). It returns True if all compared fields are equal or aborts the comparison at the first field with different values. The comparison cannot include all fields without loosing the ability of the Diff operation to provide meaningful results. The skipped fields are st_ino, st_ctime and st_blocks. As already stated in Section 5.4.3.2, neither of the first two fields can be exported at all and the value of the st_blocks field for a specific file might differ between the original image and the image exported from the Image Store.

The __eq__ method of the Node class does something very similar. First it compares its class with the class of the passed node (Line 339). If the two nodes are instances of different classes the comparison is aborted immediately. Otherwise, it iterates of its instance variables using the list in the __slots__ class variable. It returns True if all compared fields are equal or aborts the comparison at the first field with different values. Like the `__copy__` method it skips all fields having names starting with an underscore letter. This skips the `_parent` field that is different for any invocation from a `diff_to` method, but compares both the names of the nodes as well as their `Stats` instances. Furthermore, the comparison includes fields like `target` and `rdev` from the `SymbolicLink` and `Device` classes for corresponding instances. Not only does this approach skip the parent reference, but also fields that are difficult to compare, e.g., the list of children `_children` from the `Directory` class. This list cannot be easily compared, because the order of the child nodes is not relevant for equality and it is at least questionable whether a comparison should be done recursively just like `diff_to`.

The `__ne__` methods invoked for the inequality operator `!=` of both the `Stats` and `Node` class are simply negating the result of the corresponding `__eq__` methods invoked for the equality operator `==` instead of implementing extensive checks themselves (Lines 177 and 352).

5.4.5 Direct Mounts

The Direct Mount feature of the Image Store is implemented by creating a virtual file system using the llfuse [91] library, a python binding for the *Filesystem in Userspace* (FUSE) [58] library. FUSE enables implementing a virtual file system solely in user space, which significantly eases the implementation, because it can be written in Python just like the other parts of the Image Store and the existing code can be reused.

An llfuse file system implementation needs to implement an Object with a set of file system operations (implemented as methods) that correspond to functions a “real” file system, i.e., a file system implemented in the Linux kernel, provides. Examples of such operations are *open* or *read*. An excerpt of the code implementing these operations is shown in Listings 5.19 and 5.20. In addition to these operations, `llfuse` requires the use of specific data structures to pass metadata of files and request information about the implemented file system.

Listing 5.18 shows two data structures used to implement the Direct Mount feature of the Image Store. The `EntryAttributesMapper` class creates a mapping between the `Stats` instances used to store metadata in the Image Store and the `EntryAttributes` data structure that is expected by llfuse whenever metadata of files needs to be passed to the library. The latter contains all fields of the `stat` structure (see Table 5.6) except `st_dev` as well as three additional fields. All but two of the former fields are stored inside the `Stats` instances, even with the correct name. Nevertheless, the `Stats` instances cannot be used as `EntryAttributes` directly, because of the missing fields[20].

An `EntryAttributesMapper` instances stores a reference to a node – not to a `Stats` instance, because it does not contain a reference to the corresponding node. Accesses to the available fields are just delegated to the node’s `Stats` instance, as

[20] Other than that, there is no reason not to use the instances directly. The use of duck typing in Python generally facilitates such an approach.

```
23  class EntryAttributesMapper(object):
24      __slots__ = ['node']
25
26      def __init__(self, node):
27          self.node = node
28
29      def get_st_ino(self):
30          return self.node.stats.st_ino
        ...
59      def get_st_rdev(self):
60          return self.node.rdev
61
62      def get_st_blksize(self):
63          return 4096
        ...
68      st_ino = property(get_st_ino)
 ⋮      ...
81      st_rdev = property(get_st_rdev)
82      st_blksize = property(get_st_blksize)
83
84      generation = 1
85      entry_timeout = 300
86      attr_timeout = 300
 ⋮
88  class NodeEntry(object):
89      __slots__ ['node', 'attr', 'lookup_count', 'open_count', 'handle']
90
91      def __init__(self, node, inode):
92          self.attr = EntryAttributesMapper(node)
93          self.node = node
94          self.node.stats.st_ino = inode
95          self.lookup_count = 0
96          self.open_count = 0
```

Listing 5.18 Direct Mounts – Data Structures. This listing shows the data structures required for mapping between the in-memory representation of a manifest and the data structures required by the llfuse API.

shown using the example of `st_ino` (Line 30). For the two missing fields `st_rdev` and `st_blksize` appropriate implementations are provided: the `st_rdev` value is retrieved directly from the node (Line 60, whereas the `st_blksize` value is a fixed value in this implementation (Line 63). For all those fields properties are generated that allow access to the field values if they were regular fields instead of access methods (Lines 68 to 82). The three additional fields are implemented as fixed values (Lines 84 to 86).

Like a regular file system, llfuse identifies files using inode numbers. The virtual file system implementation must be able to look up the node, the `EntryAttributesMapper` as well as state information using an inode number. The `NodeEntry` class is a container to combine this information into a single entity. It is comparable to an inode in a real file system. The constructor of the `NodeEntry` class creates an `EntryAttributesMapper` instance for the passed node and assigns the node a new inode number (also passes as argument), although the latter is just an optimization and not required. The original inode numbers could be used as well.

Listing 5.19 shows the initialization of the `Operations` class that implements the file system operations. The most important fields of these class are in Lines 137 to 139: `entrycache` stores a list of `NodeEntry` instances – the inodes of the virtual file system, `hard links` is used for hard link detection in the manifest, `filecache` stores file handles of open files, and `inode_open_count` stores the number of times each file was opened. More details about the usage of the last two dictionaries can be found below in the description of the operations.

```
133 class Operations(llfuse.Operations):
134     def __init__(self, manifest, datastore):
135         self.manifest = manifest
136         self.datastore = datastore
137         self.entrycache = [ None ]
138         self.hardlinks = dict()
139         self.filecache  = dict()
140         self.highest_inode = 1
141 
142         self.fstat = llfuse.StatvfsData()
143         self.fstat.f_bsize = 4096
            ...
152         for node in manifest:
153             self.genEntryAndInode(node)
154 
155         self.hardlinks = None
        ...
167     def genEntryAndInode(self, node):
168         inode = self.highest_inode
169         new_entry = True
170 
171         if (not isinstance(node, mis.manifest.nodes.Directory) and          ↩
         node.stats.st_nlink > 1):
172             if(not self.hardlinks.has_key(node.stats.st_ino)):
173                 self.hardlinks[node.stats.st_ino] = inode
174             else:
175                 inode = self.hardlinks[node.stats.st_ino]
176                 new_entry = False
177 
178         if new_entry:
179             entry = NodeEntry(node, inode)
180             self.highest_inode += 1
181             self.entrycache.append(entry)
182         else:
183             node.stats.st_ino = inode
```

Listing 5.19 Direct Mounts – Initialization. This listing shows the initialization of the `Operations` class that creates all data structures required by the virtual file system.

The `fstat` field stores another data structure required by llfuse. It stores some metrics about the virtual file system. Most of these values are set two zero, because they are only relevant for write access, i.e., to quickly check if there are enough space and inodes left on the file system. Only the (fixed) block size of the file system (`f_bsize`) is initialized. Afterwards, the loop in Lines 152 to 153 iterates over all nodes and generates the required data structures to include the nodes in the virtual file system using the `genEntryAndInode` method[21]. When all required data structures are initialized, the `hard links` dictionary is no longer required and removed to save memory (Line 155).

The `genEntryAndInode` method is creating a `NodeEntry` instance the passed node. It includes the hard link detection algorithm already shown in Section 5.4.3.2 to ensure only a single `NodeEntry` instance is generated for multiple nodes representing a single file. Additionally, it assigns new inode numbers to the `NodeEntry` instances (Line 179) to be able to store them in a list, accessible using the inode number as index. As inode numbers start with 1, the first element in the list has to be a placeholder (Line 137).

Listing 5.20 shows a few exemplary methods implementing important operations on the virtual file system. The `readdir` method iterates of the child nodes of a directory node. It returns the name, attributes and offset (in the list of files of the directory node) of each child node. Both the child nodes attributes as well as the directory nodes itself are accessed using the `entrycache` list with the inode number as index (Lines 205 and 207).

The `lookup` method implements an operation to look up files and directories by name in a given parent directory. This method must be able to correctly handle the "`.`" and "`..`" directory entries, (Lines 212 to 216), although the current implementation of the `readdir` method does not return them for simplicity. Regular files are looked up by name using a dictionary representation of the directory nodes' list of children (Line 219). The Lines 216 and 220 are the reason for storing the generated inode number in the nodes, because in those lines the `NodeEntry` instance corresponding to a node must be found.

The `open` method opens a file for reading. Instead of keeping a list of used handles and looking up a free file handle each time a file is opened, the inode number is misused as file handle. Consequently, if a file is opened multiple times, the same file handle is returned every time. To keep track of how many times a file was opened, which is important for closing the file after usage, the open_count of the `NodeEntry` class is used as counter. If a file is opened for the first time[22], the condition in Line 270 is `True` and the underlying file in the Data Store is opened. The actual file handle to access the file contents in the Data Store is stored in the `handle` field of the `NodeEntry` class. Note that it is actually not required to store a reference to the `NodeEntry` instance in the `filecache` dictionary (Line 272), because the inode number is equal to the file handle. The use of the `filecache` has been added in preparation for switching to real file handles.

The `release` method closes an open file. The underlying file in the data store is only closed if the value of `open_count` is zero (Lines 278 to 279), i.e., `release` has been called the same number of times as `open`. Otherwise, only the counter is decreased (Line 277) and the file is kept open.

[21] Note that it is also possible to generate the required data structures on-demand instead of generating the in advance when the virtual file system is mounted. The latter approach has been chosen because it is easier to implement and to debug, but there are no technical reasons preventing the lazy approach from being implemented.

[22] Alternatively, if a file that has previously been opened was closed before the call to `open`.

```
class Operations(llfuse.Operations):
    ...
    def readdir(self, inode, off):
        for child in self.entrycache[inode].node._children[off:]:
            off += 1
            yield(child.name, self.entrycache[child.stats.st_ino].attr, off)

    def lookup(self, parent_inode, name):
        entry = None

        if name == '.':
            entry = self.entrycache[parent_inode]
        if name == '..':
            parent = self.entrycache[parent_inode].node.parent
            entry = self.entrycache[parent.stats.st_ino]
        else:
            try:
                node = self.entrycache[parent_inode].node.children_as_dict[name]
                entry = self.entrycache[node.stats.st_ino]
            except KeyError:
                raise llfuse.FUSEError(errno.ENOENT)

        entry.lookup_count += 1
        return entry.attr
    ...
    def open(self, inode, flags):
        entry = self.entrycache[inode]
        entry.open_count += 1
        if entry.open_count == 1:
            entry.handle = self.datastore.openData(entry.node)
            self.filecache[inode] = entry
        return inode

    def release(self, fh):
        entry = self.filecache[fh]
        entry.open_count -= 1
        if entry.open_count == 0:
            entry.handle.close()
            del self.filecache[fh]

    def read(self, fh, offset, length):
        if fh in self.filecache:
            handle = self.filecache[fh].handle
            handle.seek(offset)
            return handle.read(length)
        else:
            return ''
```

Listing 5.20 Direct Mounts – File System Operations (Excerpt). This listing shows the implementation of five important file system operations for reading directories, accessing files by name, as well as opening, reading from and closing files.

The last method shown is `read`. It reads at most `length` bytes starting at `offset` from the file with the file handle `fh`. The implementation of this method is straightforward. First the `seek` method is used to set the underlying file's current position. Afterwards, the `read` method is used to read the requested amount of data.

The remaining 6 methods required to provide complete read-only access to the virtual file system, i.e., `getattr`, `opendir`, `forget`, `readlink`, `statfs`, and `releasedir`, are not shown, because they are even easier to implement than the methods shown in Listing 5.20.

5.4.6 Advanced Features

In this section some advanced features of the Image Store are presented: efficient updating of virtual machines stored in the Image Store, efficient updating of exported virtual machine images, and advanced content filtering. These features are almost completely "implemented" using the features of the Image Store that have been described in the last sections: the import and export processes, filter operations, and the Direct Mount feature.

5.4.6.1 Efficient Updates

In this section four approaches are presented to accelerate the update processes for virtual machines stored in the Image Store. As already described in Section 5.3.4.5, the regular update process consists of three steps:

1. The virtual machine image is exported from the Image Store.
2. The virtual machine is updated, i.e.,
    a) The virtual machine is started on an arbitrary host.
    b) Updates are installed in the virtual machine or it is modified otherwise.
    c) The virtual machine is shut down.
3. The updated virtual machine image is reimported into the Image Store.

The update process is shown again in Figure 5.14 for easier comparison with the modified update procedures.

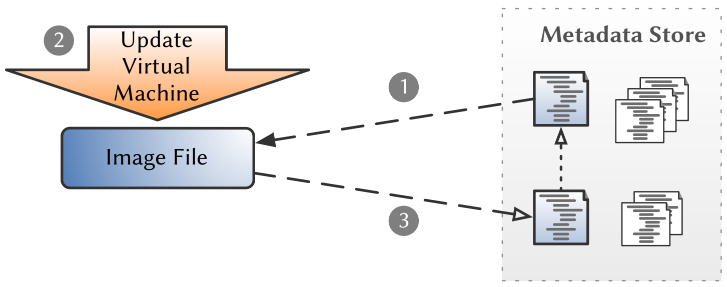


Figure 5.14 Basic Virtual Machine Image Update Process. The non-optimized process of updating a virtual machine image stored in the Marvin Image Store consists of three steps. In Step 1 the image file is exported. Afterwards, the virtual machine is updated in Step 2. Finally, the updated image file is reimported in Step 3. The dotted arrow depicts the version control feature: it points from the current to the preceding version of the manifest. (Data Store omitted.)

Both Step 1 and Step 3 of the update process can be optimized to allow efficient updating of virtual machine images stored in the Image Store. The first two approaches aim to optimize the time required for reimporting the updated virtual machine (Step 3), whereas the last two approaches aim to optimize the time required for exporting the virtual machine image (Step 1).

Diff-based Reimport

The problem with the last step of the update process is the fact that the whole virtual machine image has to be reimported, even though most of the time only a small number of files in the image were actually modified. The time needed to reimport an updated version of a virtual machine image depends on the number and the size of the files contained in the image, because for each file the metadata has to be extracted and a hash value of its contents has to be calculated. The process can thus be accelerated if only files that have actually (or very likely) been changed are imported. The first approach that addresses this problem is the *Diff-based* Reimport approach.

The update process using the Diff-based Reimport approach is identical on the update process depicted in Figure 5.14 except for the reimport step (Step 3). This reimport step can be further subdivided into four phases. Figure 5.15 depicts those four phases of the reimport step. The export and update steps are not shown.

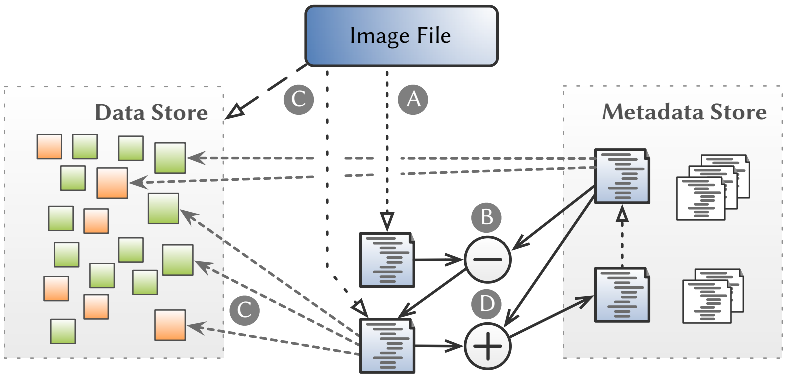


Figure 5.15 Diff-based Reimport Step. The Diff-based Reimport step consists of four phases: A) scanning of the updated virtual machine, B) applying the Diff operation to detect changes, C) importing modified files and recording their hash values in the manifest, and D) applying the Merge operation to create the manifest of the updated version of the virtual machine. Note that the manifests created in Phases A and B do not contain references to the Data Store. The references are not added until the modified files are imported in Phase C. Like in Figure 5.4, the dotted arrow in the Metadata Store points to the preceding version of the manifest.

In Phase A the entire virtual machine image is scanned. Contrary to the regular import process that extracts metadata and stores the files' contents in the Data Store in a single step, scanning the virtual machine image extracts only the metadata of the file system, but does neither calculate hash values of the files' contents nor actually store them in the Data Store. For this reason, the scan process is significantly faster than a regular import process. It is the first step in a *two-step* import process. In Phase B the differences between the manifest created in Phase A and the manifest of the virtual machine are determined using the Diff operation that is described in Section 5.4.4.3. Note that this operation works solely on the metadata recorded in the manifests. It does not even consider the hash values of files, because these values are not available in the manifest created in Phase A. The result of the Diff operation is the *update manifest*, a manifest containing all changes in the virtual machine image.

In Phase C the update manifest is used to control the second step of the two-step import process: the importing of files' contents. For every file contained in the update manifest the file's contents are stored in the Data Store and the hash value of the file's contents is recorded in the update manifest. It is important to note that the number of file nodes in the update manifest is typically much smaller than the number of files in the image. Therefore, calculating hash values selectively and storing contents only for modified files significantly reduces the duration of the import process. Finally, in Phase D the update manifest is combined with the manifest of the virtual machine using the Merge operation that is described in detail in Section 5.4.4.2. The result of this operation is the manifest of the updated virtual machine. Like the Diff operation used in Phase B, the Merge operation works solely on the metadata recorded in the manifests and thus it is very fast.

Note that it is theoretically possible that the Diff-based Reimport step might overlook a changed file. But this can only happen if a file's content is changed – without changing the file's size – and its metadata is reset to the values from before the change. In this case, the Diff operation would not be able to detect the change, because it works solely with the metadata and the hash values of the files' contents are not available[23]. Regular update installations or configuration changes – the use case for this approach – do not change the metadata in such a way that. Consequently, this problem is a theoretical one.

Layer-based Reimport

The second approach that addresses the problem of long import times is the *Layer-based* Reimport approach. It is based on the virtual machine image composition technique presented in Chapter 4. The modified update process using this approach differs from the regular update process shown in Figure 5.14 in Steps 2 and 3. It is depicted in Figure 5.16.

A layered virtual machine containing the virtual machine image to update and a temporary layer is used to conduct the update process in Step 2, after the image has been exported regularly in Step 1. The semantics of the union mount used to implement layered virtual machines ensure that all changed files are contained in the temporary layer that is called *update layer* in this use case. After the virtual machine is shut down, the Layer-based Reimport step is executed, which consists of two phases. In Phase A the update layer is imported into the Image Store using the regular import

[23] In general, the Diff operation checks the hash values of files. The unavailability of hash values is a limitation of this specific use case.

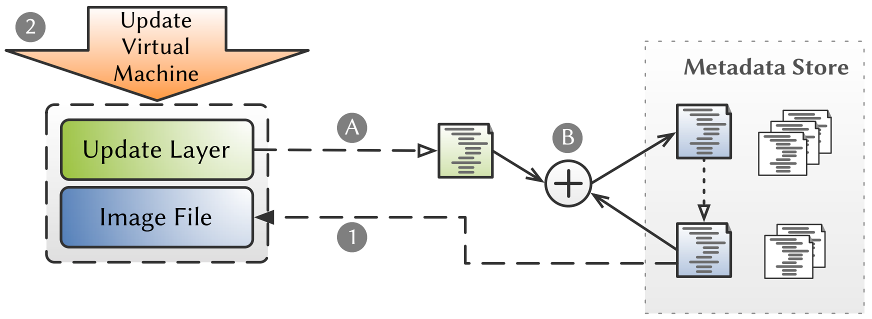


Figure 5.16 Update Process using the Layer-based Reimport. This is an improved version of the update process shown in Figure 5.14. Step 1 is unmodified. In Step 2 a layered virtual machine containing the exported image file and an update layer is started and the updates are installed. Step 3 is subdivided into two phases. In Phase A the update layer is imported and in Phase B its manifest is merged with the manifest of the virtual machine to create the manifest of the updated version of the virtual machine.

process. Afterwards, the manifest of the update layer is combined with the manifest of the virtual machine using the Merge Operation to create the manifest of the updated virtual machine in Phase B.

The advantage the Layer-based Reimport step over Step 3 of the regular image update procedure is that an update layer typically is smaller than the entire updated virtual machine image, because it only contains the changed files. This accelerates the reimport process, although the advantage depends on the size of the virtual machine image and the number and size of the updated files.

Layer-based Update Process

The remaining factor influencing the time required to update a virtual machine image is the export of its image file (Step 1). The biggest advantage of the Layer-based Reimport approach in this regard is that it can be used even if the virtual machine image that should be updated is not writable. This facilitates a range of possibilities to optimize the both Step 1 and 3 of the update process at the same time. Some of those possibilities of the `Layer-based Update Process` are described below.

In the best case, the virtual machine is already exported and kept in the image cache of an execution host. With the Layer-based Reimport approach this image file can be used for updating the virtual machine without modifying it and thus rendering it unusable for other virtual machines that depend on that specific version of the image file. It is even possible to use the image file in an update process if it is actively used by another layered virtual machine as read-only layer, which is the default for image files in the image cache (see Section 4.3). Consequently, this does not only optimize the export step of the update process, but allows skip it altogether.

Even if the exported image is not the latest version of the virtual machine available in the Image Store, this update approach can be used. However, the update layer will likely be a bit larger and thus the reimport process will need more time compared to an update layer created with an up-to-date version of the virtual machine image. The image update process described in Section 5.4.6.2 can be combined with this approach to quickly update a virtual machine image to its latest version that is betters suited for installing updates and thus smaller update layers and faster reimports.

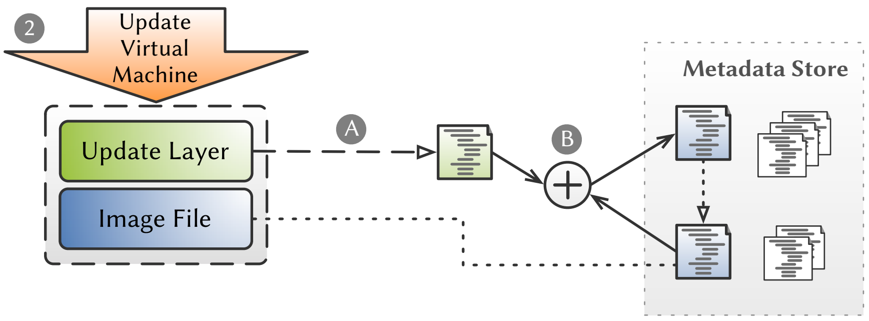


Figure 5.17 Update Process using an Existing Image File. This is an improved version of the update process shown in Figure 5.16. Exporting the image file in Step 1 of the update process is skipped altogether. The dotted line just marks that the image file corresponds to the manifest. Steps 2 and 3 are unchanged.

In the worst case, no exported image file of the virtual machine is available for use in the update process. However, the Direct Mount feature (Section 5.4.5) can be used to quickly provide a basis for installing updates as well. Together with the Layer-based Reimport approach, this allows to directly update virtual machines without exporting them beforehand. However, this *Direct Update* approach is limited to modifications of a virtual machine image that do not require the virtual machine to be running, because with the current implementation there is no way to use Direct Mounts from inside virtual machines.

If necessary, a special environment for the update step can be set up using the *chroot* system call that restricts access to a specified area [120] containing the virtual machine image and allows to execute package management utilities in the virtual machine image without booting the virtual machine. The integrity of the host operating system is somewhat protected using this approach, although ways exist to break out of such a *chroot jail* [90]. If a higher isolation between the host and the environment for the update step is required, Linux Containers (LXC) can provide a more advanced version of chroot jails with increased security for the host system. However, for simple tasks like replacing a broken configuration file a chroot environment is not required.

While this limits the usability of the Direct Update approach of virtual machines in public Cloud environments, there are nevertheless use cases for the Direct Update approach. Both in Virtualized Grid Computing and private Cloud environments the access to physical nodes required by the Direct Update approach is available at least for administrators. With physical access the administrators are able to update any virtual machine in such an environment using the Direct Update approach.

The Layer-based Update Process using a Direct Mount (Direct Update) is depicted in Figure 5.18. It is almost identical to the update process using the Layer-based Reimport step shown in Figure 5.16. There are two major changes: in Step 1 the virtual machine image is not exported, but just mounted. This saves a lot of time depending on the size of the virtual machine. Furthermore, in Step 2 the update is not installed in a virtual machine using the image composition technique. A union mount is used to combine the mounted virtual machine image and an update layer exactly like the Image Compositor does. If required, a chroot jail or a Linux Container can be used as a restricted environment for the update step. Step 3 is the usual Layer-based Reimport step.

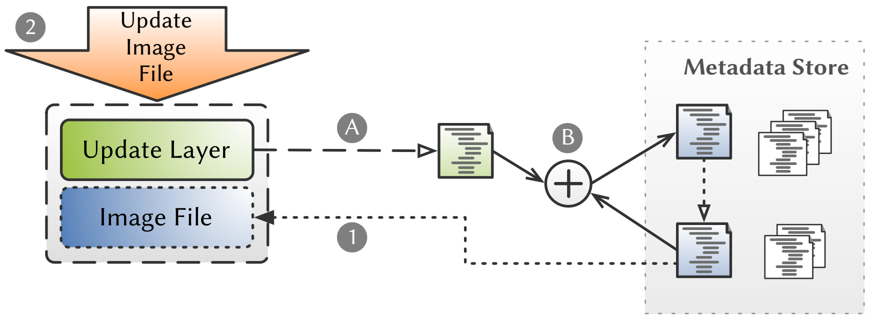


Figure 5.18 Layer-based Update Process using a Direct Mount. This is an improved version of the update process shown in Figure 5.16. In Step 1 the manifest is mounted instead of exported. In Step 2 the virtual machine image is updated using a union mount of the image and an update layer. Step 3 is unchanged.

5.4.6.2 Updating Image Files

In the last section the Merge and Diff operations have been utilize to optimize the update process of a virtual machine stored in the Image Store. The aim was to either optimize the export or the reimport process. In this section an approach for updating virtual machine image files, i.e., updating an exported image file so it corresponds to the latest version of the virtual machine stored in the Image Store, is described. If an older version of the image file is available this can significantly reduce the export time, because the amount of data that needs to be exported is reduced. Note that this approach is not limited to updating an image file to the latest version: an image file can be updated to any version of the virtual machine stored in the Image Store, even to older versions[24].

Exported images can only be updated correctly if they are still in pristine state, i.e., they were not modified after export. Unfortunately, during normal usage of image files in virtual machines the images are modified. Using the image composition approach presented in Chapter 4 it is possible to use image files as read-only layer in a composite disk image. Consequently, the image files are not modified and can thus be updated using the approach shown in Figure 5.19.

The first step of the update procedure is the use the Diff operation to compute the

[24] It is even possible to “morph” a virtual machine into another virtual machine using this approach. In this case, the advantage of this approach over a full export depends on the amount of difference between the two virtual machines.

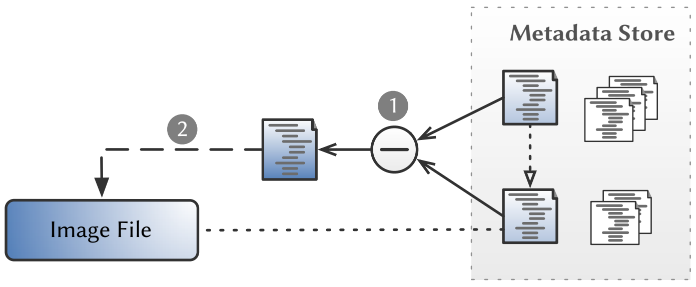


Figure 5.19 Update of an Exported Image File. An exported image can be updated in two steps. The first step is to compute the difference between the exported version and the desired version of the image using the Diff operation. In the second step the manifest containing the changed files is then exported into the existing image file instead of an empty image file. The dotted line marks that the image file corresponds to the manifest, whereas the dotted arrow points to the preceding version of the manifest.

differences between the version of the manifest that corresponds to the exported virtual machine image and the desired version the virtual machine image should be updated to. The result of the Diff operation is another manifest that contains only files, special files, and directories that have changed in the desired manifest. This manifest is then exported using the existing image file as target and a special subclass of `Exporter` shown in Listing 5.21 that implements handling of whiteout files and opaque directories specific for this use case.

```python
68  class UpdateExporter(Exporter):
69      def handleWhiteout(self, node):
70          fileops.remove(self.getPath(node))
71
72      def handleOpaqueDirectoryPre(self, node):
73          fileops.remove(self.getPath(node))
```

Listing 5.21 Exporter for Updating Exported Image Files. This listing implements a subclass of the `Exporter` class that handles both whiteout files and opaque directories in a way that enables updating virtual machine images.

The Diff operation creates whiteout nodes for every node in the reference manifest that does not have a counterpart in the current manifest. When updating an image file, this means the corresponding file needs to be deleted. The `handleWhiteout` method in the `UpdateExporter` takes care of this (Line 70). Contrary to regular directories, the content of opaque directories is not merged with the content of an underlying directory. The implementation of the `export` method in the `Directory` class, which is also used to export opaque directories, implements the merge strategy for directories: in Line 722 in Listing 5.7 the optional `keep_directories` keyword argument of the `remove` method is set to `True` to prevent the deletion of an underlying directory. By exporting its child nodes into the existing directory, it merges its contents with those of the underlying directory. To implement an opaque directory, the exporter has to delete an underlying directory before the `export` method of the `Directory` class is called. This is achieved using the `handleOpaqueDirectoryPre` hook (Line 73) that is executed before the method is called.

5.4.6.3 Advanced Content Filtering

There are two different approaches to search for and to remove unwanted or sensitive files from virtual machines: Metadata and Live Filters. The first approach works solely on the manifest of the virtual machine. It searches for files or directories that match specified patterns. This is a very fast process, since it is done completely in memory. The second approach uses a combination of image composition and the Merge operation to implement the filtering. This approach has two advantages. First, it allows the use of arbitrary tools for filtering. Furthermore, the filtering process is not restricted to deleting files, but it can also modify files contained in the virtual machine image. The filtering process can be executed inside a virtual machine to increase the security and facilitate filtering using filter scripts and tools provided by a user of the Image Store.

Metadata Filters

Metadata Filters are implemented using the Reject operation described in in Section 5.4.4.1. The filtering is typically based on patterns describing the paths files that shall be removed from the virtual machine image. A few exemplary patterns are given below:

```
/var/cache/apt/archives/**
/var/lib/apt/lists/**
/var/log/**
/tmp/**

/root/.ssh/
/root/.bash_history
/home/*/.ssh/
/home/*/.bash_history
```

A few exemplary patterns that can be used in Metadata Filters.

The first set of four patterns facilitates filtering unwanted files from Debian GNU/Linux virtual machines. These patterns remove software packages and repository databases cached by the apt package management system[25], log files as well as temporary files from a manifest. Note that these four patterns use the "`**`" wildcard, because there may be subdirectories containing files.

The last four patterns can be used to filter sensitive files, specifically the configuration directory of SSH and the history file of the bash shell for every user of the virtual machine. Note that the trailing "`/`" in the pattern describing the `.ssh` directory is

[25] Cached software packages are generally installed and can thus be deleted. Cached repository databases can also be deleted, because they are completely downloaded on every `apt-get update` to fetch the current list of available packages.

important. Without it, the directory will not be filtered, because the regular expression is anchored at the end of the string and the path of the directory node ends with a forward slash[26]. These four patterns can be combined into the two following patterns using the "∗∗" wildcard:

```
**/.ssh/
**/.bash_history
```

Combined patterns to remove sensitive files.

As already stated in Section 5.3.4.6, the filtering of unwanted and sensitive files should typically not be done in a single step, but separated from each other on different occasions. Unwanted files should be removed during the import process, before the files are copied to the Data Store, using the two-step import process first described in Section 5.4.6.1 together with the Diff-based Reimport approach. On the other hand, sensitive files should only be filtered if an image is shared, keeping the sensitive files in the Image Store to ease later usage of the virtual machine by its owner.

Figure 5.20 shows the optimal import and filtering process using Manifest Filters. In the first step, the virtual machine image is scanned as first step of the two-step import process. This extracts only the metadata of the file system, but does neither calculate hash values of files' contents nor actually store them in the Data Store. The result of the first step is the *unfiltered manifest* depicted in red. Note that this is a temporary manifest that is not stored in the Metadata Store. In Step 2, the unfiltered manifest will be filtered using a Manifest Filter (*UF*) that removes unwanted files. The result of this step is the *private manifest* depicted in yellow that does not contain any unwanted files, but might still contain sensitive files and thus should not be shared of the Image Store.

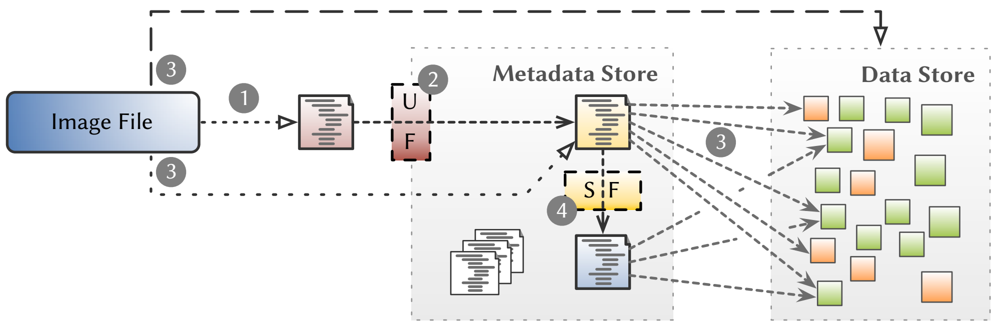


Figure 5.20 Optimal Import and (Metadata) Filtering Process. This figure shows the optimal import and filtering process using Metadata Filters: 1) the virtual machine is scanned and the unfiltered manifest (red) is generated, 2) unwanted files are filtered resulting in the private manifest (yellow), 3) contents of all files in the private manifest are stored and their hashes recorded, and 4) sensitive files are filtered resulting in the public manifest (blue).

In Step 3, the contents of all files in the private manifest are copied to the Data Store and the hash values of the files' contents are added to the private manifest. This concludes

[26] Alternatively, the anchoring can be rendered useless by appending "∗∗" instead of "/".

the two-step import process that prevents unwanted files from being copied to the Data Store. In the last step, the private manifest is filtered using another Manifest Filter (*SF*) that removes sensitive files. The result of Step 4 is the *public manifest* depicted in blue like the original virtual machine image. It does not contain sensitive files anymore[27] and can thus be shared with other users of the Image Store.

Live Filters

As described above, the idea of Live Filters is to use arbitrary tools – potentially even provided by users of the Image Store – to remove unwanted or sensitive files from virtual machines. The filtering process is very similar to the update process using an existing image file (Figure 5.17) described in Section 5.4.6.1. The Live Filter makes changes to an image file that are captured using a filter layer, which is then converted into a manifest that is combined with the manifest of the virtual machine. However, there is a peculiarity concerning the filtering of unwanted files: unwanted files stored in the Data Store at all and thus it is not possible to import the virtual machine image using the regular import approach and to remove the unwanted files afterwards using referenced the update process.

Figure 5.21 shows the optimal import and filtering process using Live Filters – the counterpart to the process shown in Figure 5.20. In the first step, the image is scanned as first step of the two-step import process. The result of this step is the *unfiltered manifest* depicted in red. In Step 2, the unfiltered manifest is filtered using a Live Filter (*UF*) that removes unwanted files. This can either be done in a virtual machine using the Image Compositor or using a union mount of the image file and a temporary layer, either directly in the host system or using a chroot jail or Linux container as proposed together with the Direct Mount update process.

Irrespective of chosen approach, in Step 3 the writable layer (*UF Layer*) that captured the changes made during the filtering process is imported using the regular import process, which copies the contents of files in the layer into the Data Store and records their hashes in the manifest. The latter is combined with the unfiltered manifest in Step 4 using the Merge operation. The result of the operation is the *private manifest* depicted in yellow. Note that at this point the private manifest only contains references to the Data Store for files that were modified during the filtering process and imported in Step 3.

In Step 5, the two-step import process started in Step 1 is finished. For every file contained in the private manifest, the file's contents are stored in the Data Store and the hash value of the file's contents is recorded in the private manifest. It is important to note that image file is sufficient for finishing the import, although any file that was changed during the filtering process is not included in the image file. The union mount of image file and UF Layer used for filtering is not required, because the contents of any changed files were already imported in Step 3. The method of the Data Store that is used for storing file contents does not calculate the hash value of a file's contents if it is already recorded in the node and returns early if the content is already stored in

[27] Assuming that the filter removes all sensitive files from the manifest.

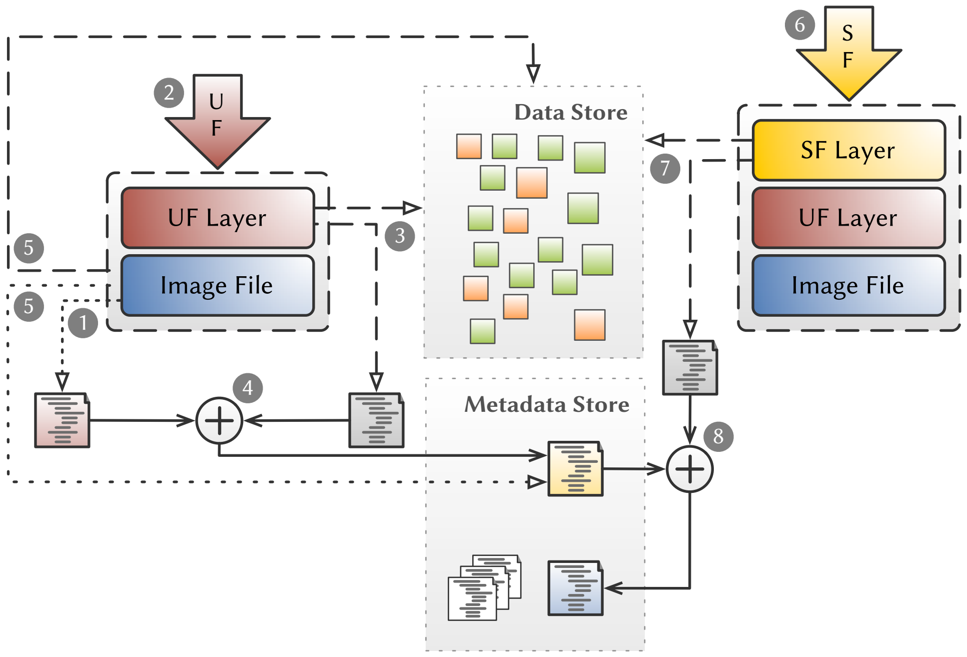


Figure 5.21 Optimal Import and (Live) Filtering Process. This figure shows the optimal import and filtering process using Live Filters: 1) the virtual machine is scanned and the unfiltered manifest (red) is generated, 2) unwanted files are filtered, 3) the filter layer is imported using the regular import process, 4) the unfiltered manifest and the manifest from Step 3 are merged to generate the private manifest (yellow), 5) the contents of all files in the private manifest are stored and their hashes recorded, 6) sensitive files are filtered, 7) the filter layer is imported using the regular import process, and 8) the private manifest and the manifest from Step 7 are merged to generate the public manifest (blue). References from manifests to the Data Store are not shown.

the Data Store (Lines 78 to 79 and Lines 84 to 85 in Listing 5.3). Therefore, the missing files are never actually accessed during the import process.

The filtering of sensitive files is done in the remaining three steps. In Step 6, the private manifest is filtered using a Live Filter (*SF*) that removes sensitive files. Like Step 2, this can either be done in a virtual machine or using a union mount of the image file, the UF Layer from Step 2 and another temporary layer. Irrespective of chosen approach, in Step 7 the writable layer (*SF Layer*) that captured the changes made during the filtering process is imported using the regular import process, which copies the contents of files in the layer into the Data Store and records their hashes in the manifest. The latter is combined with the private manifest in Step 8 using the Merge operation. The result of the operation is the *public manifest* depicted in blue. The filtering process is finished at this point.

Neither the unfiltered manifest depicted in red nor the manifests of the UF and SF Layers depicted in gray are stored in the Metadata Store. Instead, they are deleted after the filtering process is finished. Note that there are various variants of the filtering approach, e.g., in Step 6 an exported image or a Direct Mount of the private manifest can be used instead of the image file and the UF Layer, which is especially useful if the filtering of sensitive files is not done directly after the Steps 1 to 5 and either the original image file or the UF Layer are not available anymore.

The Live Filter approach is much more complex, but especially for filtering of sensitive files the flexibility of the approach with regard to availability of arbitrary tools for filtering as well as modifications of files, i.e., remove credentials from a configuration file instead of deleting it altogether, compensates for the additional complexity. However, for filtering unwanted files the use of a Manifest Filter is likely sufficient. Both approaches can be combined that way.

5.5 Experimental Results

This section contains an evaluation of the Image Store. First, the sets of virtual machines used in the measurements are described. Then, the results of an analysis of the compressing back end's efficiency for various compression levels is presented together with a list of back ends selected for further measurements. Afterwards, the different manifest on-disk formats are compared. In the main part of this section the storage and access efficiency of the Image are evaluated. The section ends with an evaluation of the Direct Mount feature and two approaches for efficient updating.

All measurements have been conducted on two identical machines containing two Quad-Core AMD Opteron 2356 CPUs running at 2.3 GHz, 16 GiB RAM, and a 500 GB SATA II hard disk. At the time of the measurements, the machines were running the Debian GNU/Linux 6.0.5 (Squeeze) operating system.

5.5.1 Virtual Machines Used in the Evaluation

Two distinct sets of virtual machines have been used for the evaluation. Set *A* consisting of 31 Debian GNU/Linux 6 virtual was also used in the evaluation of the Image Compositor and is described in Section 4.5.1.

Set *B* consists of 198 virtual machines, subdivided into six groups. Group *B1* contains the 31 regular virtual machines from Set *A*. Group *B2* contains variants of the 31 virtual machines from group *A* that where built using the same ruby program that created the original virtual machines, but using Debian GNU/Linux 7 (Wheezy) instead of Debian GNU/Linux 6 (Squeeze). Groups *B3*, *B4*, and *B5* consist of 33, 33, and 30 virtual appliances from BitNami[28] based on Ubuntu 10.10, Ubuntu 12.04, and OpenSuse 11.3, respectively. The last group *B6* contains 40 virtual machines from TurnKey Linux[29] based on Ubuntu 10.04.

An overview of the groups can be found in Table 5.11. It lists the aggregate values for all virtual machines in each group both for the number of files they contain as well as their size. The latter is considered both from the inside, i.e., the size of all files contained in an image, as well as the outside, i.e., the size of the image file. This

[28] BitNami is a provider of open source stacks [17].
[29] TurnKey Linux is a library of virtual appliances [156].

enables assessing the overhead of the file system and the image file. Note that in this case the size of the image file means its actual size on disk. The images are sparse files of up to 20 GiB apparent size. More detailed information including minimum, maximum and average values can be found in Table A.1.

Table 5.11 Virtual Machines of Set *B*. This table gives an overview of the six groups of appliances in terms of the installed Linux version, the group's size (number of virtual machines), the aggregate number of files contained in the group's image files as well as the aggregate size of the group's contents and image files, both original and optimized if the sizes are available.

			Aggregate			
				Size (MiB)		
ID	Linux Version	Group Size	Number of Files	Content	Original Image Files	Optimized Image Files
B1	Debian Squeeze	31	836,947	23,382.9	44,968.6	29,377.3
B2	Debian Wheezy	31	891,085	25,997.0	52,704.9	31,989.7
B3	Ubuntu 10.10	33	988,486	27,318.3	–	32,556.5
B4	OpenSUSE 11.3	33	1,124,348	34,195.7	–	39,393.9
B5	Ubuntu 12.04	30	952,959	34,311.8	–	38,627.8
B6	Ubuntu 10.04	40	1,251,366	25,629.9	–	28,082.0

Note that the downloaded appliances are made available in an optimized version, i.e., they have very likely been copied to a blank image file after they have been created, whereby some of the files have been omitted, e.g., temporary files, log files, or caches of the package management software. This is a reasonable decision for image files supposed to be downloaded. In typical Virtualized Grid and Cloud Computing environments the image files might not always be optimized, because this is additional step during the preparation of a virtual machine image that has to be repeated for every change of the virtual machine.

For the sets *B1* and *B2* that consist of self-made virtual machines both the original image sizes after creation as well as the optimized image sizes are recorded in the table. This optimization is however limited to copying the image to a blank image file without deleting anything. Optimization of virtual machine images is also taken into account when the Image Store is compared to other storage technologies.

Unfortunately, layered versions of the appliances of group *B3*–*B6* do not exist, so that they can be only used to evaluate the storage efficiency of the Data Store and the implemented deduplication technique for a large set of virtual machines, but not for other measurements.

5.5.2 Evaluation of Compression Algorithms

Before the Image Store is compared to other methods for storing virtual machine images, both the storage efficiency and the runtime impact of the different back ends using compression is evaluated. It is important to distinguish two different metrics to measure the size of the Data Store: the *apparent size* and the *disk usage*. The apparent size of the Data Store is the sum of the file sizes (`st_size`) of all files in the Data Store, whereas the disk usage is the sum of the number allocated 512 byte blocks (`st_blocks`) for all files in the Data Store multiplied by 512. The former is the amount of bytes that are actual content and the latter is the amount of bytes occupied in the underlying file system.

For this evaluation the largest regular virtual machine image in set *A* (*A31*) was chosen. This image file has been imported using the default back end that stores the file contents unmodified to generate a baseline value. The relative apparent size and disk usage of the Data Store as well as the relative import time of the image file for all combinations of compression levels and compressing back ends are shown in Figure 5.22. In all measurements the Data Store was empty at the beginning of the measurement. Note that there is only a value for lzma in the case of compression level 0, because there is no compression level 0 in bzip and the compression level 0 in zlib does not compress the content and thus even increases the size of the Data Store. The latter is therefore not considered anymore in the remainder of this thesis.

The apparent size of the Data Store can be reduced down to 28.0 % - 24.1 %, 22.9 % - 21.9 %, and 20.4 % - 17.7 % using the zlib, bzip, and lzma back ends with the various compression levels, respectively. The reduction is slightly lower when the disk usage is considered: 30.9 % - 27.2%, 25.9 % - 25.0%, and 23.6 % - 21.0 %, respectively. At the same time, the import time rises to 125.6 % - 732.6 %, 401.2 % - 520.5 %, and 256.5 % - 5,793.0 % of the baseline value, respectively. The import time increases almost linearly for bzip, but there are huge increases in import time for zlib with compression levels 8 or 9 and lzma with compression levels greater than 3. All results can be found in Table A.2 (in the rows with *No* in the column *4K Thr.*).

A deeper analysis of the resulting Data Stores reveals that approx. 20.5 %, 18.8 %, and 19.4 % of the files in the Data Store have a larger apparent size than their corresponding original files using all compression levels of the zlib, bzip and lzma compressing back ends, respectively. Most likely, these files were already compressed and the Data Stored compressed them again. Fortunately, the actual disk usage is only increased for approx. 0.16 %, 0.75 % and 0.10 % of the files, respectively. Detailed results of this analysis can be found in Table A.3. The reason for the very small increase of actual disk usage is that most file systems allocate space in larger blocks. In the case of the *ext4* file system used during the measurement these blocks are 4,096 bytes large. The disk usage only increases if the file size grows above the border of the last 4,096 bytes block. On the contrary, the import very likely is more affected by this, because the compression process takes longer for files that have already been compressed.

Another interesting observation is that approximately 84.4 %, 82.6 % and 83.5 % of the grown files have an original apparent size below 4,096 bytes for the zlib, bzip, and lzma compressing back ends, respectively. As 4,096 bytes is the block size of the file system, compressing such a file cannot reduce the disk usage at all. Consequently, all compressing back ends were modified so that files with an apparent size less than or equal to a threshold, i.e., the block size of the file system that contains the Data Store, are not compressed. The Data Store can determine during export whether or not a file is compressed by comparing its original apparent size with the threshold value. The original apparent size is part of every file name used in the Data Store and

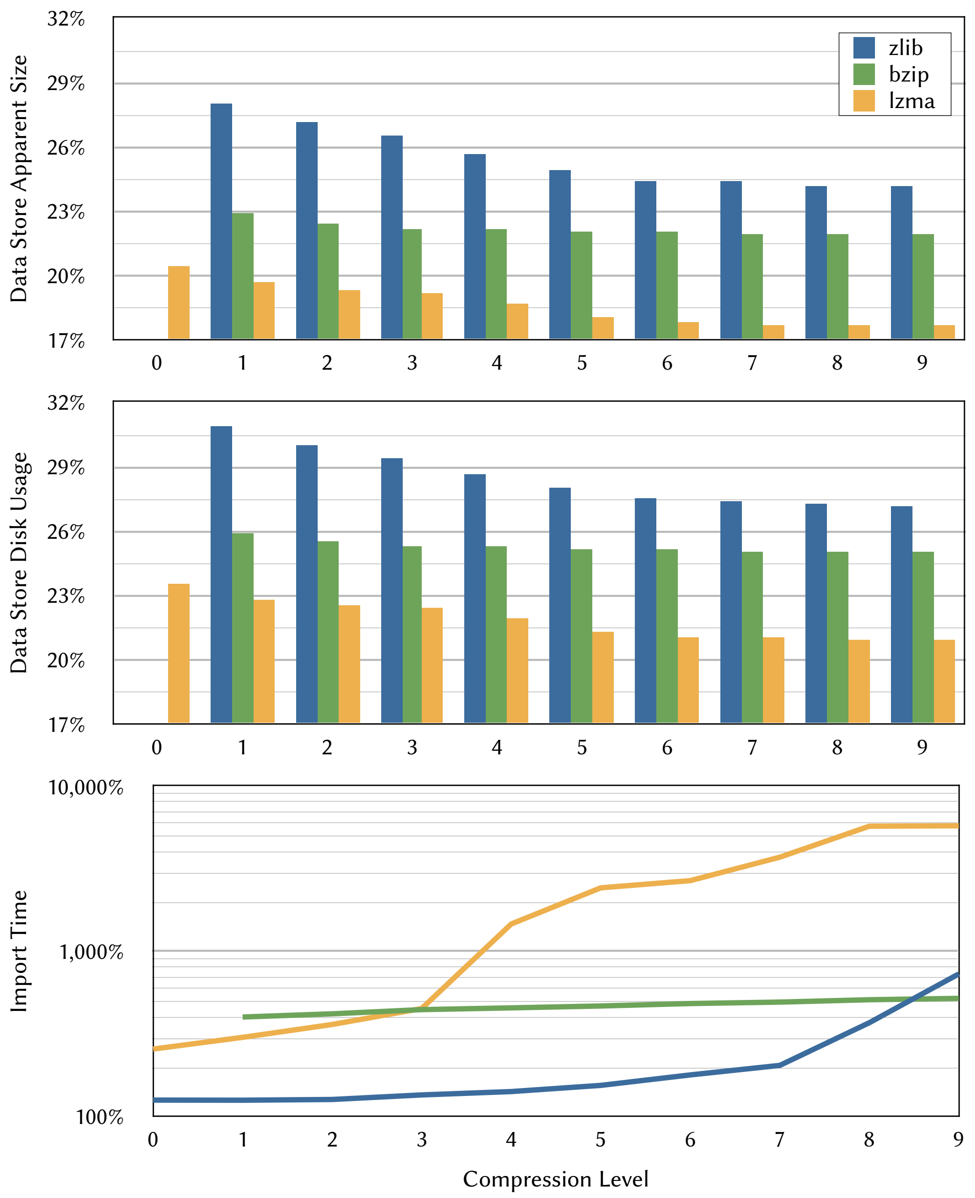


Figure 5.22 Data Store Size and Import Time. The apparent size and disk usage of the Data Store as well as the import time of image *A31* for all combinations of compression levels and compressing back ends relative to the baseline value measured with the default back end.

thus this comparison can be easily done without requiring additional metadata. The modification was simplified by the fact that all compressing back ends are derived from the `CompressingBackend` class. Its modified version is shown in Listing 5.22. All relevant methods check the original size of the file that is passed as argument (Lines 217, 220 and 224). Based on whether the size is greater than the block size or not the methods either use the functionality provided by the subclass, i.e., the `openFor-Reading`, `openForWriting` and `_decompressObject` methods, or generic code for uncompressed files.

```
class CompressingBackend(Backend):
    def __init__(self, level, blocksize=4096):
        Backend.__init__(self)
        self.level = level
        self.blocksize = blocksize

    def decompressObject(self, size):
        return None if size <= self.blocksize else self._decompressObject()

    def retrieve(self, srcpath, dstpath, size):
        srcfile = open(srcpath, 'rb') if size <= self.blocksize else self.openForReading(srcpath)
        self.copy(srcfile, open(dstpath, 'wb'))

    def store(self, srcpath, dstpath, size):
        dstfile = open(dstpath, 'wb') if size <= self.blocksize else self.openForWriting(dstpath)
        self.copy(open(srcpath, 'rb'), dstfile)
```

Listing 5.22 Implementation of the Threshold Value. Modified version of the `CompressingBackend` class with a block size threshold for compression.

The relative apparent size and disk usage of the Data Store as well as the relative import time of the image file *A31* for all combinations of compression levels and compressing back ends with a block size threshold are shown in Figure 5.23.

The relative apparent size of the Data Store is increased by 0.4 % - 0.5% for all compression back ends with a block level threshold. At the same time, the disk usage of the Data Store is even slightly decreased by 0.01 % - 0.04 % with the threshold. The reason for this effect are already compressed files with an original apparent size barely below the 4,096 byte block boundary that grow to over 4,096 bytes after being compressed again. Since the back ends with the block size threshold do not compress those files, the number of allocated blocks is reduced. On the whole, the addition of the threshold does not increase the disk usage of the Data Store. All results can be found in Table A.2 (in the rows with *Yes* in the column *4K Thr.*).

On the other hand, the threshold has a substantial effect on the import time of the image. It rises to 122.4 % - 727.4 %, 392.8 % - 510.1 %, and 243.7 % - 3,711.4 % of the baseline value for the zlib, bzip and lzma back ends, respectively. Especially the import times for the lzma back end are significantly reduced by up to 36 % with the threshold, whereas the import times for the zlib and bzip back ends are reduced by up to 5 % and up to 3 %, respectively. The reduction of the import times is shown in Figure 5.24.

A deeper analysis of the resulting Data Stores reveals that the implementation of the block size threshold reduced the number of files with a larger apparent size compared to their corresponding original files size down to approx. 3.2 % - 3.3 %. The number of files with an increased disk usage is reduced to approx. 0.03 %, 0.17 % and 0.02 % for the zlib, bzip and lzma back ends with the threshold, respectively. This is shown in Figure 5.25 for the zlib back end with compression level 1. Detailed results of this analysis can be found in Table A.4.

The back ends with the block size threshold were used for all further measurements,

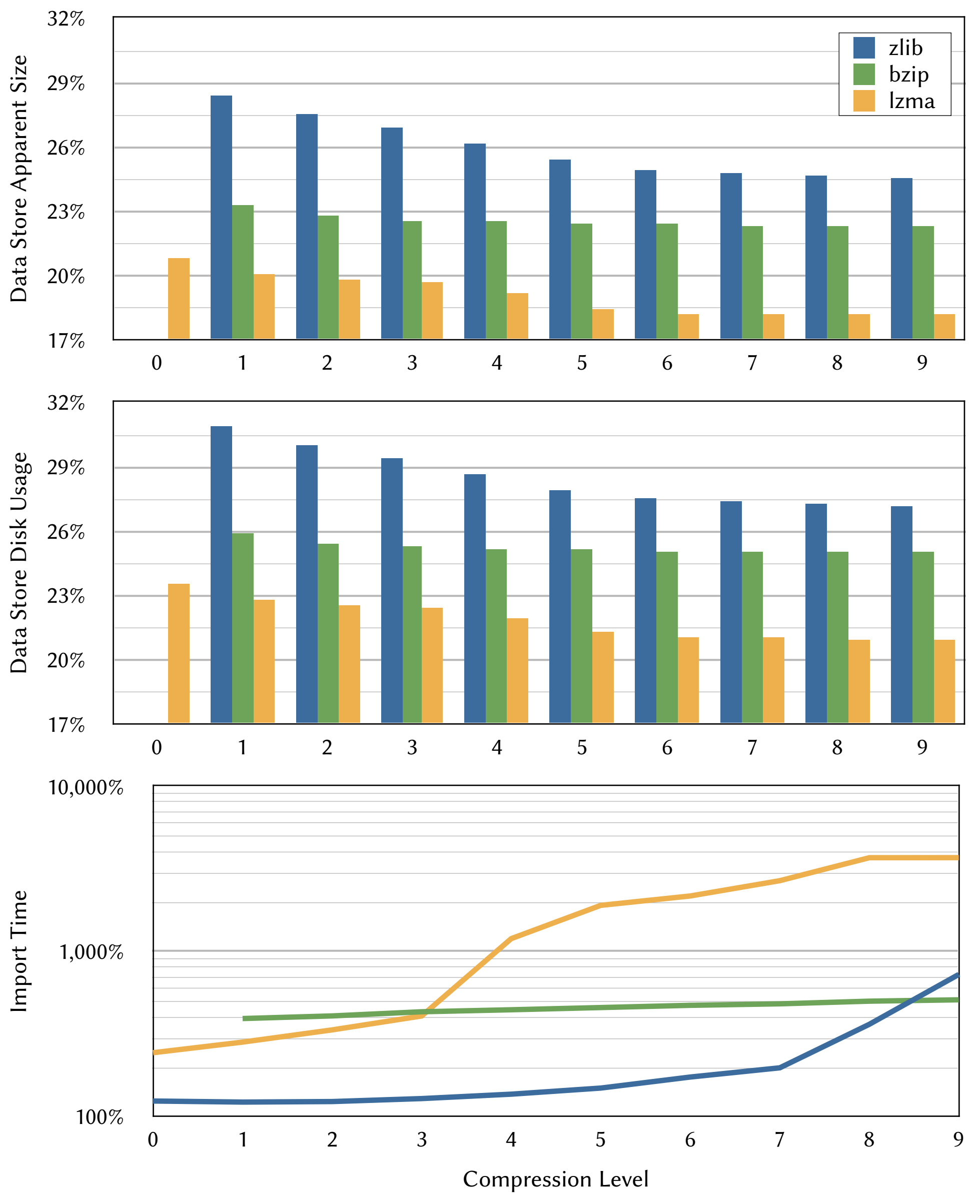


Figure 5.23 Data Store Size and Import Time. The apparent size and disk usage of the Data Store as well as the import time of image *A31* for all combinations of compression levels and compressing back ends with a block size threshold relative to the baseline value measured with the default back end.

because they do not increase the size of the Data Store and at the same time reduce the import time.

One goal of this subsection is to identify a few compression level and back end combinations that are used in the remaining chapter to further evaluate the Image Store. The selected combinations should make a compromise between Data Store size and import time. Thus, the relative import time was restricted to 300 % of the import time using the default back end. This restriction completely eliminates bzip that has

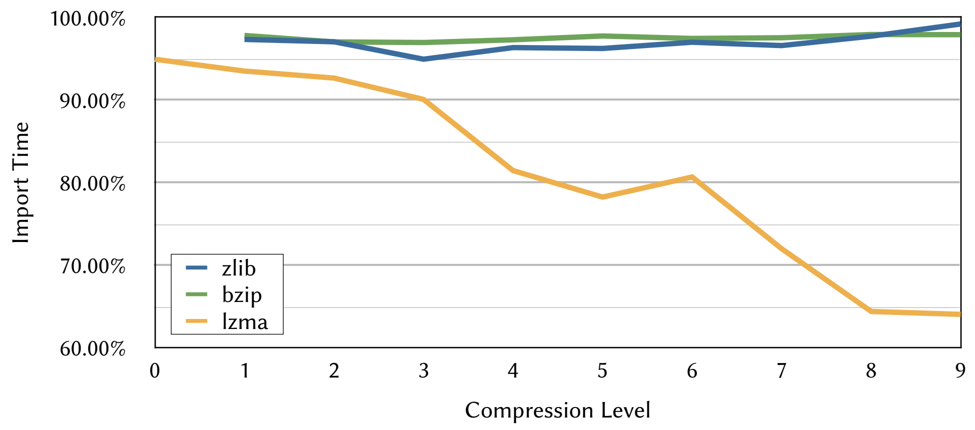


Figure 5.24 Import Time Reduction Caused by the Threshold. The reduction of the import time as a consequence of adding a block size threshold in the compressing back ends.

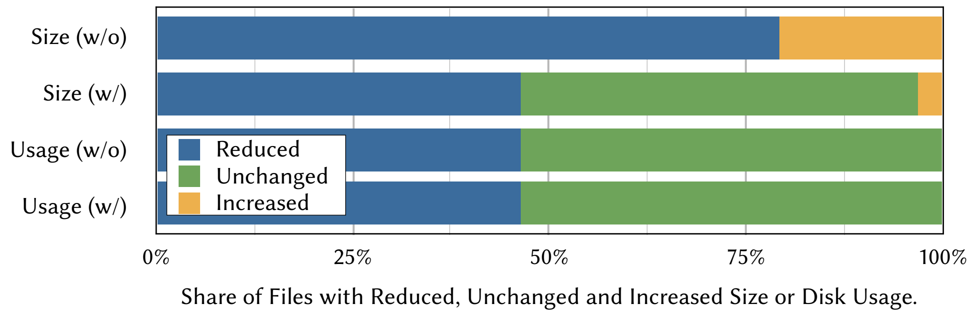


Figure 5.25 Block Size Threshold and File Size Changes. Share of files with reduced, unchanged and increased apparent size and disk usage in the Data Store using the example of the zlib back end with compression level 1 both without and with the block size threshold.

a relative import time of at least 401.2 %. Additionally, the first three levels of bzip are performing worse than lzma both in terms of Data Store size and import time. The first useful compression level of bzip is level 4 that has a relative import time of 455.9 %.

The restriction to 300 % relative import time reduces the number of combinations to nine: zlib with levels 1 to 7 and lzma with levels 1 and 2. Considering the difference in import time and Data Store size between the different levels of zlib the levels 1, 3, and 5 have been selected as well as level 1 of lzma.

5.5.3 Manifest Formats

Not only the Data Store and the used back ends are important for the storage and access efficiency of the Image Store, but also the format the manifests are stored in. In this sections the 6 combinations of the two formats XML and pickle, the pretty printing option (*pp*) for the XML format and zlib compression option (*zlib*) using compression level 1 are assessed. This assessment uses the regular and layered virtual machines in set *A*.

The combined (apparent) size of all regular and layered manifests for the images in set *A* is shown in Figure 5.26. As expected the XML format results in a very high manifest size, especially if the pretty printing option is used to improve readability. The reason is the general verbosity of XML in combination with the high number of nodes (between 330,000 and 900,000 nodes for the regular images in set *A*). On the other hand, the average size of the 16.6 MiB and 24.3 MiB for regular and 6.6 MiB and 9.8 MiB for layered images in the XML and the pretty printed XML format, respectively, are not much considering the difference between image size and content size (see Tables 4.8 and 4.9). Additionally, the verbosity of the XML format is easily compressible down to less than 10 % of the corresponding uncompressed format. The compressed manifests can still be easily accessed using tools like zcat.

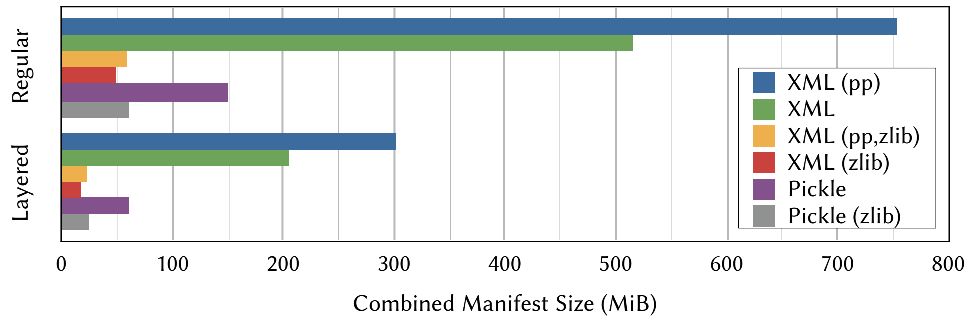


Figure 5.26 Combined Manifest Size for Set *A*. This chart shows the combined manifest size for all regular and layered virtual machines in set *A* for the different combinations of manifest formats and options.

The pickle format is better suited for the amounts of data a manifest needs to store. Even without compression the pickle format reduces the combined manifest size down to around 20 % or 30 % compared the XML and the pretty printed XML format. Compression can further reduce the combined manifest size, although the reduction is smaller compared to the XML format.

Figure 5.27 shows the average time required to read and write the manifests of the virtual machines in set *A*. It is immediately obvious that the decision between those two formats is not straightforward. The XML format can be written efficiently, but parsing the manifests in XML format is way more time consuming. Contrary, the pickle format can be read very efficiently, but writing the manifest takes significantly longer compared to the XML format. Note that the compression of the XML format is cheap in terms of runtime, although it significantly reduces the size of the manifests. This is probably due to using compression level 1 and the good compressibility of the XML format.

In the rest of this chapter, the compressed XML format without pretty printing is used in all measurements, because it provides a good trade off between storage and access efficiency. For a productive use of the Image Store the pickle format might be the

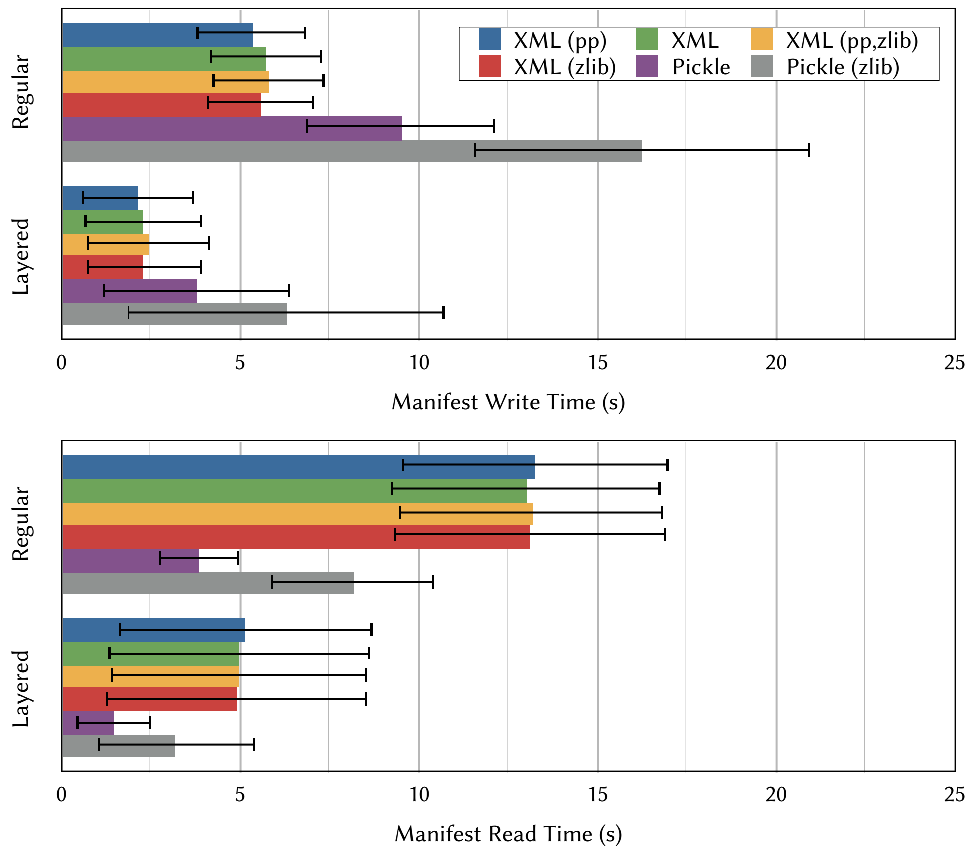


Figure 5.27 Average Manifest Read and Write Times. Average time to read (lower chart) and write (upper chart) the manifests of both the regular and the layered virtual machines in set *A* including the standard deviation.

more reasonable choice, because read access to manifests is more likely during normal usage and the pickle format is better suited for this use case.

5.5.4 Storage Efficiency

The storage efficiency of the Image Store is evaluated for the selected back ends using the two sets of virtual machines in several measurements. The first set of measurements evaluates the size of individual Data Stores for virtual machines, whereas the second and third set of measurements evaluate shared Data Store for all virtual machines in Sets *A* and *B*.

5.5.4.1 Set *A* – Individual Data Stores

Figure 5.28 shows the disk usage for each of the 31 regular and layered images in set *A* stored in individual Data Stores compared to traditional images, optimized images, compressed images as well as optimized and compressed images. The compressed images were compressed using zlib at compression level 1. The exact numbers can be found in Tables A.5 and A.6. An enlarged section of the chart for regular virtual machine images is shown in Figure 5.29, because the disk usage values for the optimized and compressed images and the compressed Data Stores are very close to each other. All disk usage values for the Image Store are including the size of the corresponding manifests in compressed XML format that lie between 1.0 MiB and 2.6 MiB for regular and between 87.3 KiB and 1.6 MiB for layered virtual machine images.

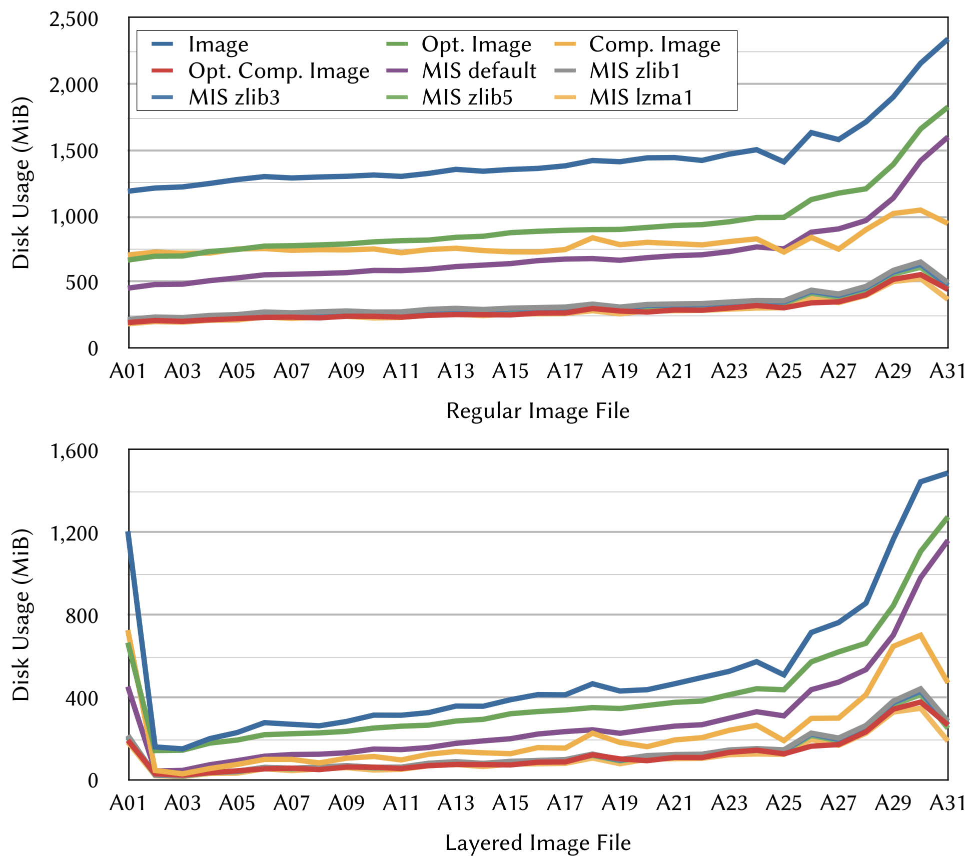


Figure 5.28 Disk Usage of Individual Images of Set *A*. Disk usage of the virtual machine images in set *A* as image, optimized image, compressed image, optimized and compressed image as well as stored in the Image Store with different back ends (*MIS* ...). An individual Data Store is created for each virtual machine in the Image Store. The upper chart shows the disk usage for regular images, the lower chart for layered images.

Even without compression the Image Store can reduce the disk usage to 38.3 % - 68.3 % and 68.1 % - 78.6 % compared to images and optimized images, respectively. The reduction is caused solely by discarding the surrounding file system. When a compressing back end is used the disk usage is reduced to 18.3 % - 31.0 % and 27.2 % - 35.4 % compared to images and optimized images, respectively. The good results of compressed images in comparison with the Image Store was expected, although it should be noted that the default back end without compression can keep up with compressed images that were not optimized beforehand except for the largest images. Images that have been optimized before compression achieve disk usage reductions

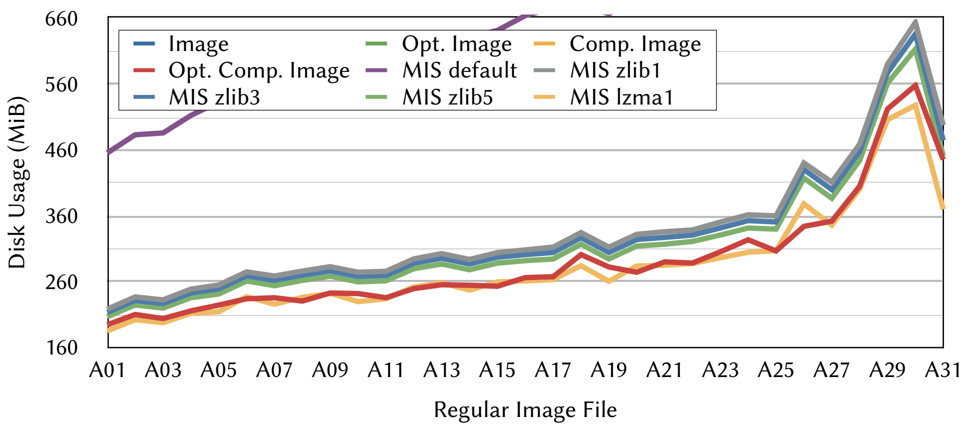


Figure 5.29 Disk Usage Storage of Individual Images of Set *A*. Enlarged section of the chart for regular virtual machine images in Figure 5.28. This chart focuses on the optimized and compressed images and the compressed data stores that are difficult to distinguish in the original chart.

comparable to the Image Store with the lzma 1 back end and better than all selected zlib back ends.

However, the comparison with compressed images is not entirely fair for two reasons: the Image Store cannot benefit from the deduplication if only a single virtual machine image is stored in the Data Store and compressed images are inferior to storage with regard to manageability and access. For example, to mount an image and check its contents, e.g., to verify if it contains specific files, a compressed image needs to be completely uncompressed first, whereas the Image Store allows to mount the image directly – depending on the use case it might even be sufficient to check the manifest only. The storage efficiency for multiple files within a single Data Store is below.

Furthermore, the measurement shows that the disk usage reductions determined in Section 5.5.2 for the different compressing back ends based on image *A31* are not achieved for all virtual machines in set *A*. Image *A31* seems to contain a lot of files that can be compressed very successfully. For regular virtual machines in individual Data Stores the disk usage is reduced to 47.4 %, 46.2 %, 44.8 % and 40.1 % of the disk usage using the default back end for the zlib1, zlib3, zlib5 and lzma1 back ends, respectively. The values for layered images are almost identical.

5.5.4.2 Set *A* – Common Data Store

After looking at the storage efficiency of the Image Store for single virtual machine images in a Data Store its efficiency for the complete set *A* – both for regular and layered images – is evaluated. The disk usage of the Data Stores created by the different back ends are compared to the same four techniques as above: images, optimized images, compressed images, as well as optimized and compressed images. Contrary to the measurements above, a single Data Store (per back end) is used for all 31 images and thus the deduplication is kicking in. The results of this evaluation are shown in Figure 5.30.

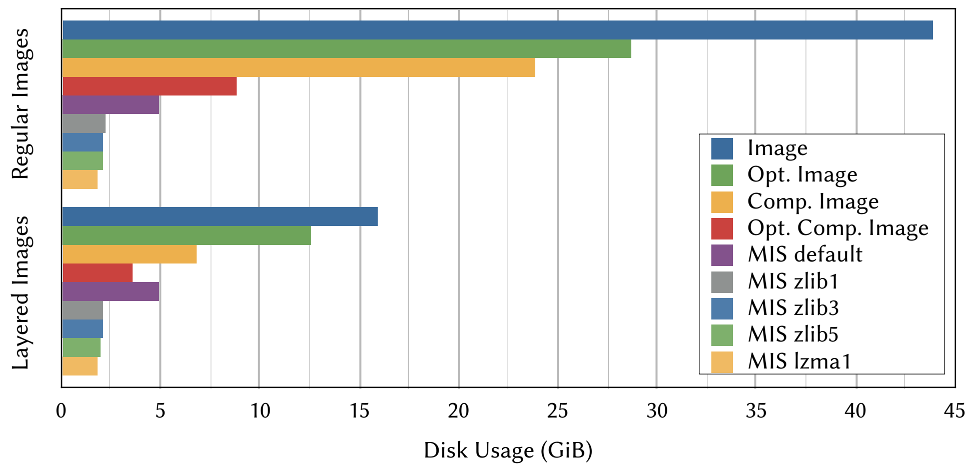


Figure 5.30 Disk Usage of All Images of Set *A*. Disk usage for all virtual machine images in set *A* as images, optimized images, individually compressed images (zlib level 1), optimized and individually compressed images, as well as stored in the Image Store with the selected back ends. A single Data Store is created for all virtual machines in the Image Store.

The disk usage of all 31 images in set *A* is significantly reduced when the Image Store is used. The default back end reduces the disk usage to 11.3 % and 17.3 % compared to the image and optimized image, respectively. With compressing back ends, the disk usage is reduced further to 4.1 % - 4.9 % and 6.3 % - 7.6 %, respectively. Equally well results are achieved compared to compressed images: 20.9 % and 7.6 % - 9.1 % for the default and the compressing back ends, respectively. The relative disk usage compared to optimized and compressed images is a little bit higher: 56.6 % and 20.5 % - 24.7 % for the default back end and the compressing back ends, respectively.

For layered images the relative disk usage is a bit higher. Using on of the compressed back ends, the disk usage is reduced to 11.1 % - 13.4 % and 14.2 % - 17.1 % compared to the images and optimized images, respectively. With the default back end, the disk usage drops to 31.1 % and 39.9 %, respectively. Compared to compressed images the disk usage falls to 73.4 % and 26.3 % - 31.8 % for the default and the compressing back ends, respectively. Only the optimized and compressed images can compete with the Image Store using the default back end. The disk usage of the Data Store is 137.9 % of the disk usage of the optimized and compressed images. However, using the compressing back ends the disk usage of the Data Store falls to 49.5 % - 59.7 % of the images' disk usage.

The compressing back ends reduce the disk usage of the Data Store containing all 31 regular virtual machine images of set *A* down to 43.6 %, 42.4 %, 40.9 %, and 35.9 % of its size using the default back end. These values are better than the values determined for virtual machine images in individual Data Stores above. Again, the values for layered images are almost identical.

5.5.4.3 Set *B* – Common Data Store

Finally, the disk usage for storing all virtual machines in set *B* is evaluated. The results of this evaluation are shown in Figure 5.31 for images and the Image Store using the selected back ends. As already said above, no layered images exist for the subsets *B2* - *B6*, so this evaluation is limited to regular images. Additionally, for the subsets *B3* - *B6* only optimized versions of the images are available. The values for images in this evaluation thus refer to the optimized versions of all images, including subsets *B1* and *B2* that are available both as optimized and non-optimized versions.

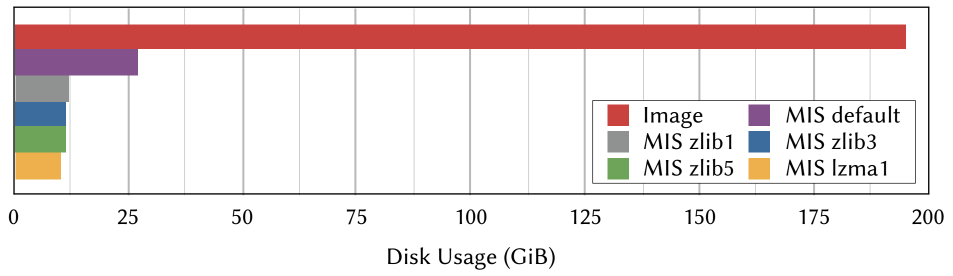


Figure 5.31 Disk Usage of All Images of Set B. Comparison of disk usage for all virtual machine images in set *B* as image files and stored in the Image Store using the selected back ends.

The Image Store reduces the disk usage for all 198 images in set *B* to 5.1 % - 6.1 % compared to optimized images using a compressing back end and 13.9 % using the default back end. The disk usage reduction is even higher compared to non-optimized images. Based on the optimization results for the subsets *B1* and *B2* the estimated total disk usage for non-optimized versions of all images in set *B* is between 260 GiB and 380 GiB. Compared to these estimated values the Image Store reduces the disk usage down to 3.9 % - 4.7 % and 2.7 % - 3.2 % for compressing back ends and 7.3 % - 10.7 % for the default back end, respectively.

The compressing back ends reduce the disk usage of the Data Store containing all 198 regular virtual machine images of set *B* down to 43.8 %, 42.9 %, 41.5 %, and 37.0 % of its size using the default back end. The reduction of disk usage achieved by the compressing back ends increased just marginally compared to the values determined above for the 31 regular virtual machine images of set *A* (*B1*).

Figure 5.32 shows the increase in total disk usage for consecutively imported subsets of set *B* compared to the disk usage of the imported virtual machines' image files. The Data Store is empty before subset *B1* is imported. It can be seen that the disk usage of the Image Store grows very slowly compared to image files.

Figure 5.33 shows the additional disk usage for each of the subsets that are imported consecutively to an initially empty Data Store using the selected back ends. Although the total (optimized) image size per subset lies between 27.4 GiB and 38.5 GiB, the additional disk usage for this set lies only between 0.5 GiB and 3.5 GiB for the compressing back ends and between 1.9 GiB and 7.0 GiB for the default back end.

Notably, subset *B4* has by far the lowest additional storage requirements when being

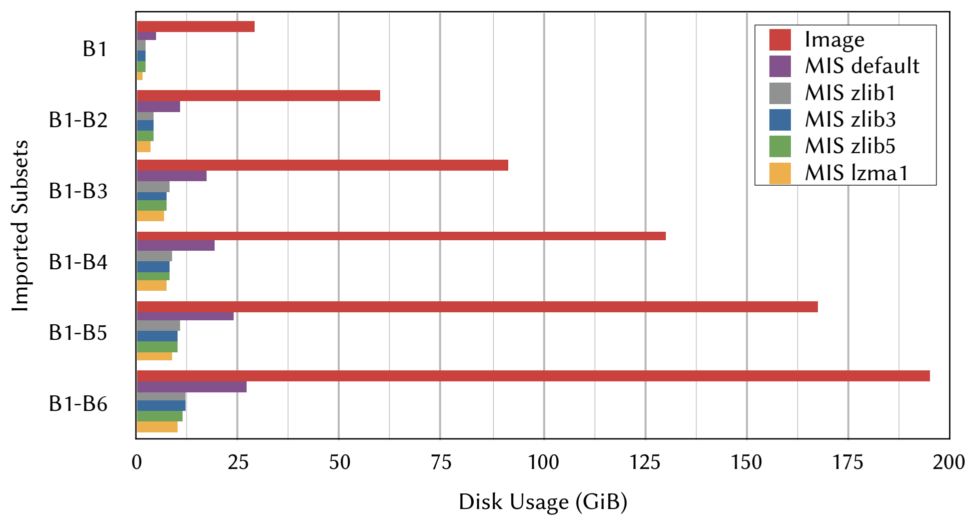


Figure 5.32 Total Disk Usage of Subsets of Set B. Disk usage for subsets of set *B* that are consecutively imported into the Image Store with the selected back ends compared to the disk usage of the corresponding image files.

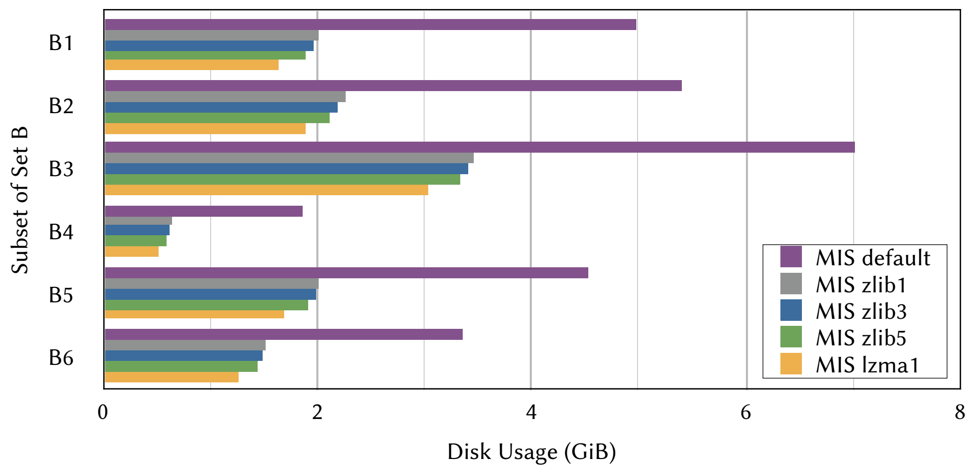


Figure 5.33 Additional Disk Usage for Subsets of Set B. Additional disk usage for consecutively imported subsets of set *B* when they are imported into the Image Store with the selected back ends.

imported into the Image Store already containing the subsets *B1* - *B3*, although *B4* is the second largest subset both in terms of content size as well as number of files. Additionally, subset *B4* is not a derivative of Debian/GNU Linux – contrary to Ubuntu. This shows that the Image Stores can efficiently store virtual machine images of different distributions and versions.

5.5.4.4 Discussion

Overall, the results of these evaluations show that the Image Store provides a very good storage efficiency, not only compared to image files, but also to their optimized and compressed counterparts. The only approach that comes close to the Image Store with regard to storage efficiency are optimized, compressed images. This is not only the case for large numbers of virtual machines such as for the entire set *B* with almost 200 virtual machines, but even for a single virtual machine stored in the Image Store.

5.5.5 Access Efficiency

The access efficiency of the Image Store is evaluated in several measurements. The first set of measurements compares the import and export times of the Image Store with three different reference values. For these measurements the regular and layered virtual machines in set *A* where used in the non-optimized versions. An individual Data Store is used for each of virtual machine images. The reference values are the following:

Image Copy — Image copy denotes copying the virtual machine image using standard tools, e.g., `cp`. This is expected to be much faster than importing, because it just copies the image like any large file without looking at its contents.

Content Copy — Content copy denotes a recursive copy of the image's contents, i.e., mounting the image file and copying all files, e.g., using `cp -r`. This process is more comparable to the import process and therefore is a better reference value, because in both cases each file in the image needs to be accessed and copied individually. Additionally, this reference value allows to distinguish between the overhead caused by fundamental approach of separating the data from the metadata as well as performing deduplication on the data and the additional overhead caused by the compressing back ends.

Compression — Compression denotes compressing the virtual machine images with zlib using compression level 1 (like in Section 5.5.4).

5.5.5.1 Import of Virtual Machines

The import times for all 31 regular and layered virtual machine images in set *A* are shown in Figure 5.34. The exact numbers can be found in Tables A.7 and A.8. The charts clearly show the expected overhead for all Image Store import processes in comparison with copying an image for regular images and for the last dozen layered images. Notably, for all but the last half dozen regular and layered images the import process using the default back end is faster or at least on par with image compression. This also partially applies to the zlib back ends.

For easier comparison of the values, the average import times for all 31 regular and layered virtual machine images in set *A* and their standard deviation are shown in Figure 5.35.

As expected, importing regular images into the Image Store is much slower compared to copying of image files. The import process needs on average 318.6 %, 380.9 % - 427.2 %, and 804.7 % of the time to copy an image using the default, zlib and lzma back ends with the selected levels, respectively. In comparison with the copying the

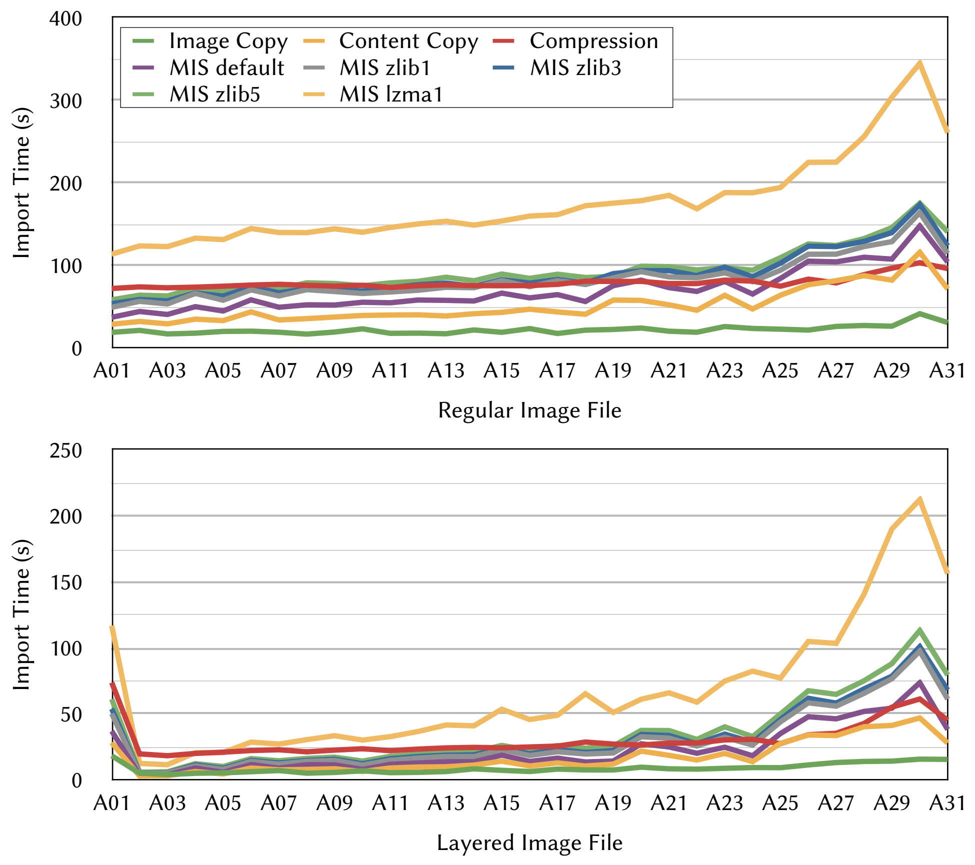


Figure 5.34 Individual Import Times for Set *A*. Individual import times for all regular (upper chart) and layered (lower chart) virtual machine images in set *A* in comparison with the reference values.

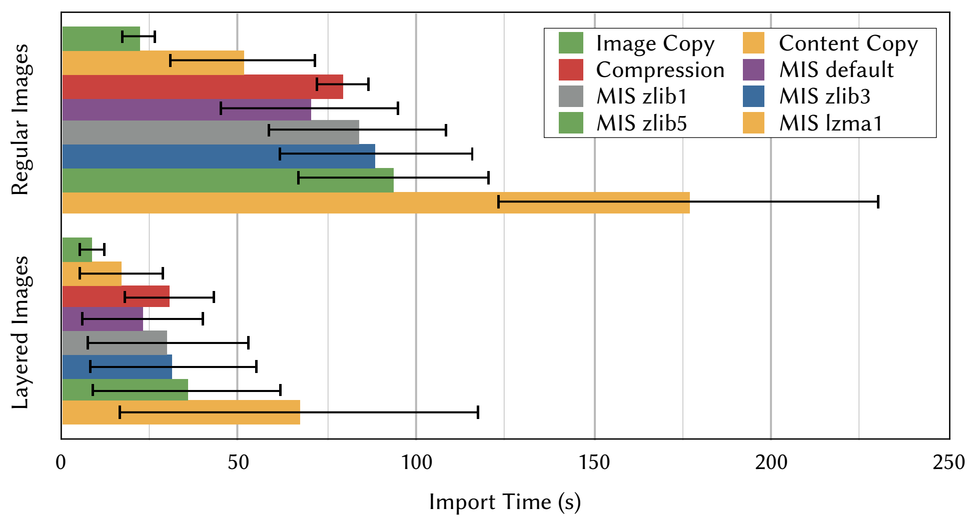


Figure 5.35 Average Import Times for Set *A*. Average import time and standard deviation both for the regular and layered virtual machine images in set *A* in comparison with the reference values.

contents of image files the Image Store performs considerably better. The import process needs 136.7 %, 163.4 % - 183.2 %, and 345.2 % of the time to copy the contents of an image using the default, zlib and lzma back ends, respectively. The Image Store import process even outperforms the compression of images with 88.1 % using the default back end, is almost on par using the zlib back ends with 105.4 % - 118.2 % and noticeably slower only using the lzma back end with 222.6 %.

The results for layered image files are comparable, although there are deviations in both directions. The import of a layered virtual machine in set *A* takes on average 259.9 %, 341.0 % - 404.5 %, and 768.8 % compared to the time to copy an image, 133.5 %, 175.1 % - 207.7 %, and 393.8 % compared to the time to copy the contents of an image and 75.2 %, 98.6 % - 117.0 %, and 221.8 % compared to compressing an image for the default, zlib and lzma back ends.

Up to now import times of virtual machine images were compared with copy and compression operations on the same images, strictly separating between regular and layered images. This enables assessing the different back ends with regards to write performance and overhead introduced by the Image Store in general. However, it does not examine the synergy of combining the Image Store with image composition: image composition allows to reduce the size of images by introducing layers and a reduced image size lessens the time required to import an image.

Consequently, the most important comparison is between the image copy of a regular virtual machine and the import process of the corresponding layered virtual machine. For the average import times shown in Figure 5.35, the relevant parts of the chart are the topmost bar in the regular images group and the five bars at the bottom of the layered images group. The import process using the default back end on average is almost on par with copying the corresponding regular image file with 103.6 % of the copy time and just moderately slower using the zlib back ends with 135.9 % - 161.2 %. Only using the lzma back end the import time on average is still significantly higher compared to copying the corresponding regular image file with 305.6 %.

For the individual import times, this comparison is depicted in Figure 5.36. It is immediately visible that the import process for the Image Store using the default and zlib back ends is on par with copying the image file for the first half of set *A*[30]. Using the default back end the Image Store can even compete with copying images for all but the last half dozen images. Only when the lzma back end is used importing images is slower than copying images for all but two images.

5.5.5.2 Export of Virtual Machines

The export times for all 31 regular and layered virtual machine images in set *A* are shown in Figure 5.37. The exact numbers can be found in Tables A.9 and A.10. Like for the import process there is a visible overhead for using the Image Store compared to copying plain images, but it is smaller especially for the lzma back end that is on par with the zlib back ends for decompression.

[30] This consideration is deliberately ignoring the base image *A01* that is naturally one of the biggest images in set *A*. Regarding its size it would have to be arranged next to *A29*.

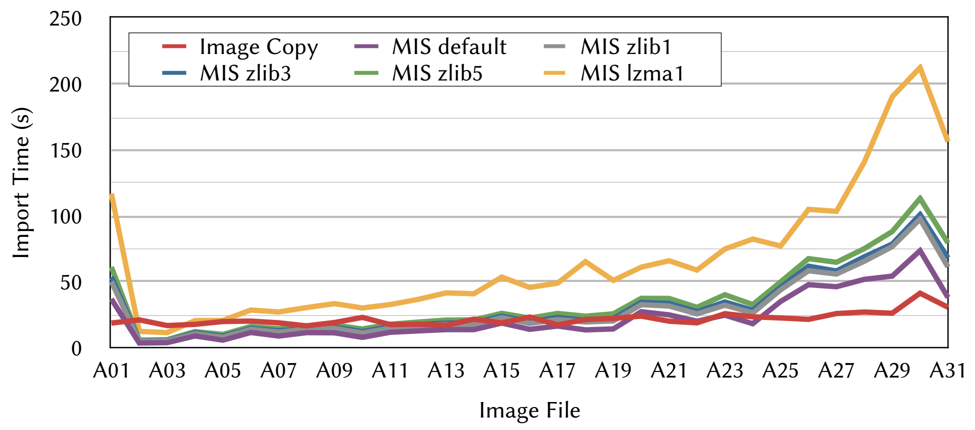


Figure 5.36 Layered Import Time vs. Regular Copy Time. Comparison of the time required to copy a regular image and to import the corresponding layered image into the Image Store using the selected back ends.

The charts clearly show the expected overhead for all Image Store import processes in comparison with copying an image for regular images and for the last dozen layered images. Notably, for all but the last half dozen regular and layered images the import process using the default back end is faster or at least on par with image compression. This also partially applies to the zlib back ends.

For easier comparison of the values, the average export times for all 31 regular and layered virtual machine images in set *A* and their standard deviation are shown in Figure 5.38.

As expected, exporting regular images from the Image Store is much slower compared to copying of image files. The export process needs on average 257.1 %, 331.6 % - 320.9 %, and 334.2 % of the time to copy an image using the default, zlib and lzma back ends with the selected levels, respectively. Note that the decompression is faster for the higher levels (this is the case for all measurements in this section). In comparison with the copying the contents of image files, the Image Store performs considerably better. The export process needs only 110.3 %, 140.9 % - 136.1 %, and 148.1 % of the time to copy the contents of an image using the default, zlib and lzma back ends, respectively. The values are slightly worse in comparison with decompressing an image file. The image store on average needs 117.9 %, 152.0 % - 147.1 %, and 153.2 % of the time required for decompression for exporting an image from a Data Store using the default, zlib and lzma back ends, respectively.

The results for layered image files are comparable, although there are deviations in both directions. The export of a layered virtual machine in set *A* takes on average 230.4 %, 274.4 % - 264.9 %, and 288.4 % compared to the time to copy an image, 118.3 %, 140.9 % - 136.1 %, and 148.1 % compared to the time to copy the contents of an image and 101.7 %, 121.1 % - 116.9 %, and 127.3 % compared to decompressing an image for the default, zlib and lzma back ends.

Like for the import process, the comparison between the image copy of a regular virtual machine and the export process of the corresponding layered virtual machine

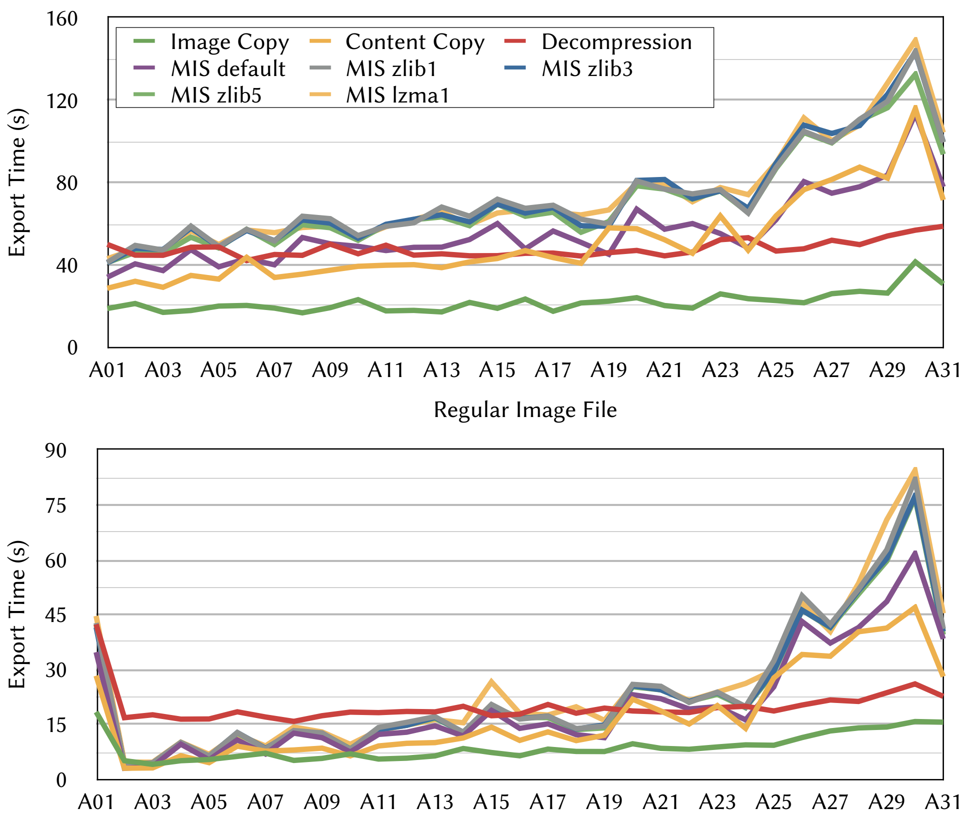


Figure 5.37 Individual Export Times for Set *A*. Individual export times for all regular (upper chart) and layered (lower chart) virtual machine images in set *A* in comparison with the reference values.

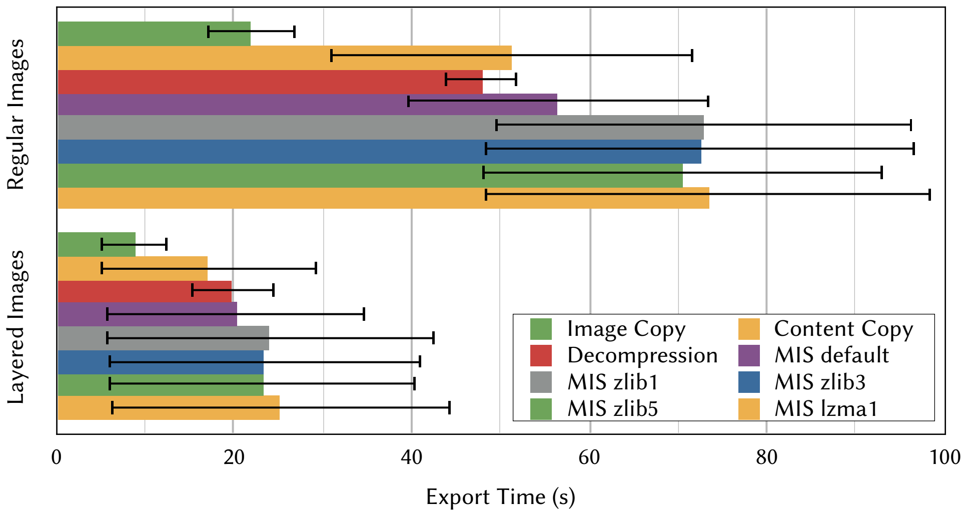


Figure 5.38 Average Export Times for Set *A*. Average export time and standard deviation both for the regular and layered virtual machine images in set *A* in comparison with the reference values.

is the most important comparison. For the average export times shown in Figure 5.38 the relevant parts of the chart are the topmost bar in the regular images group and the five bars at the bottom of the layered images group. The export process using the default back end on average is faster than copying the corresponding regular image file with 91.8 % of the copy time and just moderately slower using the zlib and lzma back ends with 109.4 % - 105.6 % and 115.0 %, respectively.

For the individual export times, this comparison is depicted in Figure 5.39. It is immediately visible that the export process for the Image Store using the default and the compressing back ends is on par with copying the image file for all but the last half dozen images in set *A*[31]. Contrary to the import process, the lzma back end is not significantly slower than the other compressing back ends when exporting images.

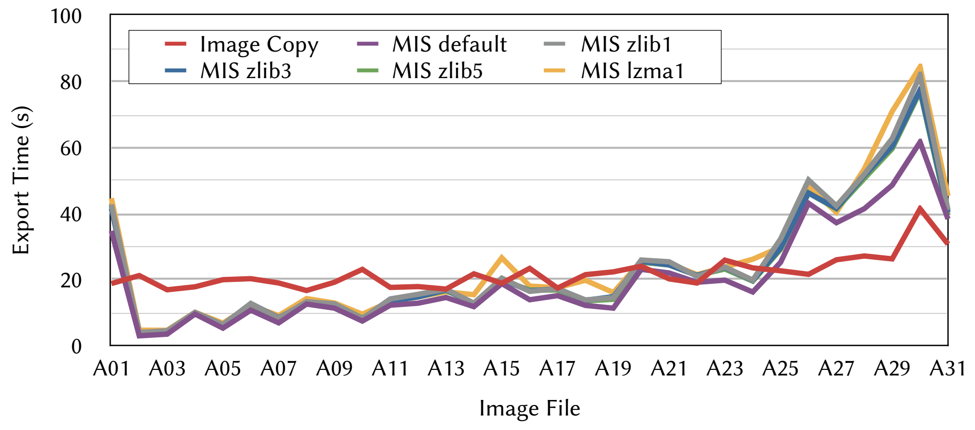


Figure 5.39 Layered Export Time vs. Regular Copy Time. Comparison of the time required to copy a regular image and to export the corresponding layered image from the Image Store using the selected back ends.

5.5.5.3 Discussion

Certainly, the extraction of metadata and content from virtual machine images during import takes longer than copying the image file. This also applies to the recreation of a virtual machine image from scratch. However, the comparison with plain virtual machine images is unfair considering the storage efficiency of both approaches. Compared to optimized, compressed images – the second best approach after the Image Store with various back ends in terms of storage efficiency – the import and export times are on par for a large portion of set *A*.

If the Image Store is combined with the Image Compositor, the results are way more in favor of the Image Store. For all but a half dozen of the virtual machines of set *A*, importing and exporting a virtual machine is faster or on par with copying the corresponding virtual machine image. For the export, this applies to all selected back ends, whereas for the import this applies only to the zlib back ends.

Access efficiency is more than just importing and exporting images, but also access to

[31] Again, the base image *A01* is deliberately ignored in this consideration.

metadata of virtual machines and content of individual files. In this regard, optimized, compressed images are out of competition, because both require a virtual machine image to be fully decompressed first. Decompressing a virtual machine in set *A* takes on average 47.9 seconds, even if only a small file should is accessed or the contests of a directory are listed. Using the Direct Mount feature, the contents of virtual machines stored in the Image Store are immediately available.

5.5.6 Advanced Features

In this section, two of the advanced features of the Marvin Image Store have been evaluated: mounting and efficient updating of virtual machines.

5.5.6.1 Direct Mounts

The performance of the Direct Mount functionality is evaluated by copying the Linux kernel sources using two different origins: an exported image and an image mounted using the Direct Mount feature of the Image Store. The Linux kernel source used in this measurement consists of 37,264 files with a combined size of roughly 500 MiB.

The Image Store was used with three back ends in this measurement, whereby two of them are not among the selected ones: the sparse back end and the zlib back end with compression level 6. The sparse back end can be considered roughly equivalent to the default back end and the zlib6 back end can be considered as roughly equivalent to the zlib5 back end. Figure 5.40 shows the amount of time required to copy the kernel sources from each of the origins.

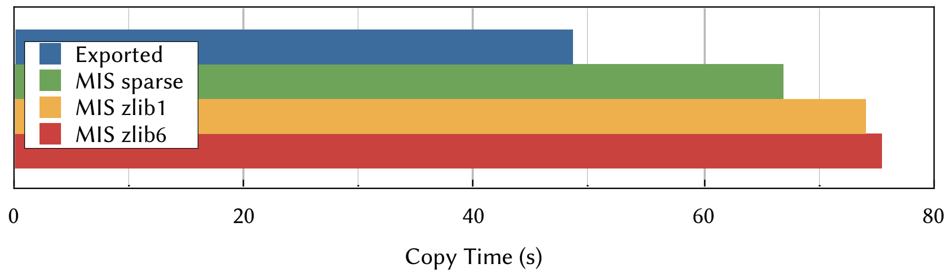


Figure 5.40 Time Required to Copy the Linux Sources. The time required to copy the Linux kernel sources from an exported image and from an image stored in the Image Store that is made available via the Direct Mount feature.

The time required to copy the entire Linux kernel sources from the stored image is 137.9 % of the time to copy from the exported image using the sparse back end. The copy time increases to 152,8 % and 155.4 % for zlib1 and zlib6 back ends, respectively. While this result seems to make this function unusable at first sight, one has to consider that the export process can be skipped using a Direct Mount, saving up to 114 seconds in case of the regular virtual machine image *A30*. Thus, a Direct Mount can still be used for maintenance operations, e.g., the efficient update process evaluated below.

5.5.6.2 Efficient Updates

In the following, the update process of a virtual machine is evaluated for three different approaches. The first approach is the obvious one: export the image, update the virtual machine and reimport it into the MIS. The second approach is update process using the Layer-based Reimport described in Section 5.4.6.1. The updates are installed into an update layer that is combined with the exported image using the image composition technique. The typically smaller update layer is then reimported and the resulting manifest is merged with the original one. The third approach is a variant of the second one in which the image is not exported at all, but mounted using the Direct Mount feature of the Image Store. Contrary to the former two approaches, the updates where not installed in a virtual machine, but in a chroot jail due to technical limitations of the implementation.

The virtual machine that is updated is neither contained in set *A* nor *B*. It is a Debian/GNU Linux 6 virtual machine with an optimized image size of 665.8 MiB. The update includes 50 packages with a total package size of about 58.1 MiB and an installed size of 337.3 MiB. Note that during this update more than half of the virtual machine image is modified. Figure 5.41 shows the update times for all three approaches using the three back ends.

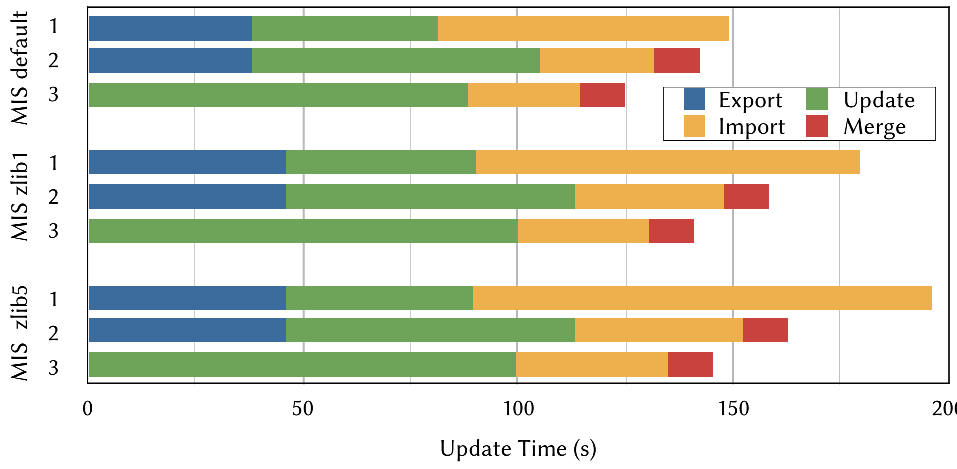


Figure 5.41 Updating a Stored Virtual Machine. Updating a virtual machine stored in the Image Store using three different approaches: 1) export, update and import. 2) export, update with update layer, import update layer, and merge. 3) mount (not shown), update with update layer, import update layer, and merge.

Although the use of an update layer increases the time required to install the updates because of the union mount used by the image composition, the savings during the import of the update layer compared to the import of the entire updated virtual machine image make the second approach preferable over the first one. Compared to the first approach, the overall time required for the update process is reduced to 95.4 %, 88.4 %, and 82.9 % for the default, zlib1 and zlib5 back ends, respectively. Furthermore, the second approach is getting better for bigger image sizes and smaller update sizes. The specific case examined here is not very typical with regard to the update size in relation the image size. Regular virtual machine updates tend to generate smaller updates and thus to increase the advantage of the Layer-based Reimport approach.

Despite the obviously much higher time required to install the update and its technical limitations the third approach using a Direct Mount instead of exporting the is even faster than the second approach. Compared to the first approach, the overall update time is reduced to 83.6 %, 78.6 %, and 74.1 % for the default, zlib1 and zlib5 back end, respectively.

5.6 Summary

In this chapter, the Marvin Image Store, a novel repository for storing large numbers of Linux-based virtual machine images efficiently, has been presented. It is based on the idea of separating the data in the image file (contents of files) from the metadata (directory hierarchy, file names, and attributes) when a virtual machine is copied into the Image Store (imported) and storing both in an appropriate way. Data and metadata are stored in a content-addressable storage system with optional compression (Data Store) and as easy accessible manifests, respectively. , i.e., in a content-addressable storage using optional compression. Image files are automatically recreated when a virtual machine is copied out of the Image Store (exported). This approach does not only increase the storage efficiency, but also facilitates novel ways to deal with virtual machine images by providing version control and a set of operations that work on the metadata. Additionally, several optimizations of the import and export processes have been proposed to further increase the access efficiency of the Image Store.

Several measurements with up to 198 virtual machines have shown that the Image Store is able to reduce the disk usage down to 4 % of the size of regular virtual machine images without spending an excessing amount of time for compression of the data. Even for a single image, the disk usage can be reduced down to 18 % of the image file's size. In combination with the Image Compositor presented in Chapter 4, around 80 % of the virtual machines[32] can be imported into and exported from the virtual machines in less than or the same amount of time compared to copying the corresponding regular machine image.

[32] Only 31 of the 198 virtual machines used in the measurements are available both as regular and layered versions. The percentage relates to the 31 virtual machines.

This page is intentionally left blank.

"Security is always going to be a cat and mouse game because there'll be people out there that are hunting for the zero day award, you have people that don't have configuration management, don't have vulnerability management, don't have patch management."

Kevin Mitnick (1963–)

6 Virtual Machine Security

6.1 Introduction

As already described in Section 5.1, there is a trend towards increasing numbers of virtual machines. This increase is driven by various reasons: very competitive prices for virtual machine usage, either on demand in the Cloud or on a subscription basis at traditional providers' computing centers, easy to use virtual machine management systems that allow the creation of a virtual machine with a few clicks, software provisioning based on virtual appliances that bundle an application with a preconfigured operating system environment, as well as the increased isolation between processes when they are executed in separate virtual machines. This increase of the number of virtual machines, a phenomenon called virtual machine sprawl [128], makes the task of keeping the software stack up-to-date even more time-consuming.

The propagation of virtual machine based computing technologies also brings along a new type of users without experience in systems management. Nevertheless, these users administer virtual machines used in Virtualized Grid and Cloud Computing environments. An example of this type of users are scientists from the computational sciences, who move to Cloud Computing from well-administered Grid Computing or local cluster environments, to get instantaneous access to a huge number of virtual machines when the local resources are not sufficient.

The combination of a large number of virtual machines and potentially inexperienced users leads to an increased risk of insufficiently maintained virtual machines. In Virtualized Grid Computing, even a single vulnerable virtual machine might endanger the whole Grid site, because once compromised it can be used as a springboard to attack other systems. Unless there is a strong isolation between the networks the virtual machines are connected to, this risk also exists for Cloud Computing. Even when the isolation between virtual machines is perfect, vulnerable virtual machines pose a risk, because an attacker can create high system load and excessive I/O and thus adversely affect other users of the same infrastructure.

Another challenge is created by the ability of users to clone virtual machines to create new ones, snapshot the current state of virtual machines or even rollback virtual machines to a previous state. While these features provide great flexibility, they also pose an enormous security risk – both for users and providers. A virtual machine rollback, for example, could reveal an already fixed security vulnerability, and cloning an existing virtual machine that contains an unfixed vulnerability can spread the vulnerability even further [59]. Especially the latter issue is very serious, because of the way new virtual machines are typically created using virtual machine management systems: by cloning a template image instead of creating the new virtual machine from scratch.

Especially in Virtualized Grid and Cloud Computing environments another key issue arises: virtual machines are likely dormant for some periods of time, depending on the usage model (Section 3.3). These virtual machines cannot be easily kept up-to-date during these phases, because typically this would require the virtual machines to be started, updated and shut down again, which is not only time- and resource-consuming, but may also be a tedious process especially if large numbers of virtual machines have to be maintained.

Generally, computers exposed to the Internet are at constant risk of being attacked. To protect them against security attacks, all security related incidents should be detected by monitoring system behavior. To detect security anomalies, Intrusion Detection Systems (IDS) or Intrusion Prevention Systems (IPS) are typically used; their combination is known as Security Information and Event Management (SIEM). However, most SIEM systems only monitor events on the infrastructural layer, need human assistance in case of error recovery, raise a high number of false alarms, and do not scale well with an increasing number of events.

Improving the security of virtual machines is an essential part of building secure environments for Virtualized Grid and Cloud Computing. Four proposals are introduced in this chapter that work hand in hand to improve the security of virtual machines in all important phases of their lifecycle and thus the overall security of Virtualized Grid and Cloud Computing environments.

The first two proposals are targeting the continuous maintenance phase: their goal is to help keeping dormant virtual machines up-to-date. This is a two-step process: first, outdated software in virtual machines has to be identified, and second, the affected virtual machines have to be updated. The first proposal is a solution called *Update Checker* that scans virtual machines for outdated software in an efficient manner, which is a challenging task especially for dormant virtual machines. The second proposal is a concept for centrally updating affected virtual machines using the image composition technique presented in Chapter 4. A set of tools is provided to support the centralized updated process.

Unfortunately, keeping the software installed in a virtual machine up-to-date is not enough. It is also necessary to check the configuration of the installed software in an additional step to detect misconfigurations or insecure services that endanger the security of the virtual machine. One option to do this is to use the *Online Penetration Suite* that is the third proposal in this chapter. It analyzes the security of virtual machines using multiple vulnerability scanners and is able to detect the issues just mentioned. The findings of the scanners are aggregated in a combined security report.

Although continuous maintenance phase can account for a very large part of a virtual machines lifetime in the on-demand execution model, the need for virtual machine security is by no means limited to this phase, but also prevalent in other phases of the lifecycle of virtual machines. The Online Penetration Suite also targets the deployment phase in addition to the continuous maintenance phase. The last proposal in this chapter is the concept of a security monitoring system for virtual machines that also covers the execution phase. The system is able to detect, analyze, and handle security anomalies including both known and yet unknown security vulnerabilities.

The phases in the lifecycle of a virtual machine that are related to the security proposals in this chapter are shown in Figure 6.1.

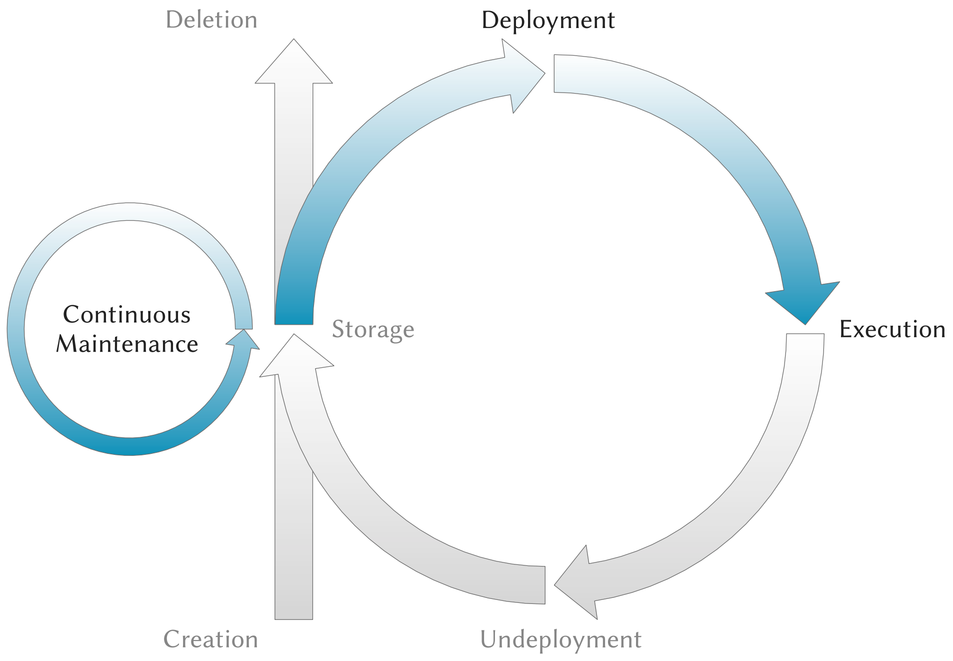


Figure 6.1 Concerned Virtual Machine Lifecycle Phases. The primary focus of virtual machine security regarding the lifecycle of a virtual machine is the continuous maintenance phase. The deployment phase is affected as well.

Parts of this chapter have been published in [141, 143, 144, 15].

6.2 Related Work

6.2.1 Update Checker

The Cloud Computing risk report written by ENISA [41] mentions the failure of customer hardening procedures as one of the research problems that needs to be solved. Customers failing to secure the computing environment may pose a vulnerability to the Cloud infrastructure.

Automation of system administration, including system administration and updating systems is one of the relevant research topics mentioned in the Expert Group Report [87] created by the European Commission.

An image management system, called Mirage, is presented by Wei et al. [167]. Mirage addresses security concerns of a virtual machine image publisher, customer and administrator. To reduce the publisher's risk, an access control framework regulates the sharing of virtual machines images. Image filters remove unwanted information, e.g., logs, sensitive information, etc., from images prior to publishing. The authors also present a mechanism to update dormant images and apply security updates. While Mirage offers a complete solution for virtual disk image maintenance, it lacks the features presented in this paper. Mirage cannot show whether the packages in a system are outdated and work with multiple package management systems.

Based on Mirage, Reimer et al. [128] present the Mirage image format (MIF), a new storage format for virtual machine disk images. It solves the problem of *virtual machine image sprawl*, i.e., the complexity of maintaining disk image content that changes continuously due to cloning or snapshotting. MIF stores the disk image content in a central repository and supports searching, installing and updating applications in all images. By using a special storage device, disk images share common blocks and thus take up only a fraction of the actual disk space. Using MIF it is also possible to update packages on a system although the update procedure is quite complex. At first, it is quite unclear how the system determines whether there is a need for an update. Furthermore, the system needs a modified version of dpkg, thus, it is not usable with off-the-shelf installations or other package management solutions. The authors state that "the optimized dpkg does not support some of dpkg's features".

A system for unscheduled system updates, called AutoPod, was presented by Potter et al. [119]. AutoPod is based on system call interposition and the `chroot` utility and is able to create file system namespaces, called pods. Every process in a pod can be offline-migrated to another physical machine by using a checkpoint mechanism. Unfortunately, AutoPod is bound to Debian GNU/Linux and cannot be used with other package managers. Furthermore, it also updates a system automatically, which could lead to problems in case of an incomplete update. In contrast to the presented solution, AutoPod is based on `chroot`, which is known for having several major security flaws in the past.

Sapuntzakis et al. [131] developed a utility, called the Collective, which assigns virtual appliances to hardware dynamically and automatically. By keeping software up-to-date, their approach prevents security break-ins due to fixed vulnerabilities. While their approach allow updating whole virtual machine appliances, it does not allow the update of certain packages within the appliance. Furthermore, it is not possible to determine whether certain packages are outdated.

Canonical, the company behind Ubuntu Linux, offers a commercial product called Landscape [24]. Landscape can be used to manage Ubuntu (virtual) machines, including package management and monitoring. While Landscape is able to detect and update outdated applications within virtual machines, it can only handle the Debian package format and is not able to update dormant machines. However, Landscape can update outdated machines once they are live the next time.

6.2.2 Centralized Update Process

There are several papers addressing storage virtualization to handle the problem of storing a large number of virtual machine images efficiently. Typically, the use of snapshots (persistent views of a virtual machine image at specific points in time) as the base for the creation of new virtual machines is proposed. *VirtuaLinux* [7] uses the Enterprise Volume Management System (EVMS) [42] as its storage. If a snapshot used as base of some virtual machine images needs to be updated, all virtual machine images based on it need to be recreated, making this approach unusable for the centralized update process.

Parallax [166, 99] uses a custom mechanism for storing virtual machine images and creating snapshots. Template images are used to build new virtual machine images that share common blocks. 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 virtual machines, Sapuntzakis et al. [132] propose a similar concept. Virtual machine images are built from a hierarchy of disks that are combined using block-oriented COW techniques at runtime. Updating individual disks from the hierarchy is intended, but requires recreation of all disks based thereon. Again, this technique is not usable for the centralized update process.

A completely different approach is used in *Ventana* [116]. Instead of virtual machine images as a virtual counterpart to physical discs, views of a virtual file system are used. A view is a combination of one or more branches that are trees of files and directories. Applying security updates to the virtual machines is not addressed in the paper.

XenoServer [80] uses NFS to access the root file system in the virtual machine, which is provided by another virtual machine called Stacking COW server running at the same host. The file system consists of a local template and one or more virtual machine specific layers called overlays that are stored remotely and accessed via the Andrew File System (AFS). Again, updating virtual machines is not addressed at all.

A proposal of virtual machines for distributed workstations that can be used as Condor nodes or virtual cluster has been made by Wolinski et al. [170]. Besides features like automatic network configuration, IP over P2P, etc. the paper also introduces a layered file system based on UnionFS. 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. However, the authors do not mention the potential problems resulting from the exchange of a layer that are described in Section 6.3.2.1.

6.2.3 Online Penetration Suite

SAVEly, a tool to check Amazon Machine Images (AMIs) for vulnerabilities was presented by Bleikertz et al. [19]. The authors construct an attack graph based on the security polices used in Amazon's EC2. These policies are used to group machines while restricting the communication between them. Based on the graph, the authors use the OpenVAS scanner to check the AMI for remote vulnerabilities. Their approach is tightly coupled to Amazon's EC2 and cannot be used with other IaaS implementations or in Virtualized Grid environments.

Yoon and Sim [177] present an automated network vulnerability assessment framework. It uses a combination of a scan manager, message relay server and scanners to check the hosts in a network for vulnerabilities. Their approach uses similar techniques as the ones presented, but it lacks the ability to work in a Cloud Computing environment. It is neither able to control virtual machines, nor to instrument an IaaS solution like the XGE.

6.2.4 Security Monitoring

Teixera et al. [153] present Holmes, an implementation of a monitoring solution for integrating a CEP engine with machine learning. The CEP engine generates alerts using hand-crafted continuous queries to detect known abnormalities and deviations from expected behavior. Furthermore, it normalizes the asynchronous events for analysis with the machine learning algorithm, i.e., it joins different streams to be analyzed together and generates time series with equidistant intervals. A machine learning algorithm detects unknown anomalies in time series, without manual rule creation and anticipation of problem conditions and thresholds.

Holmes utilizes infrastructure level sensors and can thus only detect hard- and software issues as well as attacks such as Distributed Denial-of-Service (DDoS) attacks. The proposed architecture is not hierarchical, i.e., it consists of a single message bus, where all sensors publish their information to and the central CEP engine and machine learning modules subscribe to. This architecture does not scale well, neither for an increasing number of events nor for an increasing number of machines to monitor. Historical data is not used for anomaly detection, which limits the potential to detect anomalies as well as increases the risk of false positives.

Ficco [45] presents an approach to detect and respond to attacks by using event correlation. The approach is described using a DDoS attack as an example. Different information sources on several architectural levels, such as network, operating system and application, are deployed in strategic points of the system. In the example, these sources are the number of connections from a single IP, the length of the backlog queue of TCP and the number of application requests. Agents deployed together with the sensors analyze, filter, normalize and forward messages to the so-called Decision Engine, consisting of a correlator, a diagnoser and a reaction module. Specialized modules, called remediators, are used to remediate a specific attack or intrusion. An ontology is used to map all symptoms and possible effects of an attack. This ontology is used for the correlation of events and the decision about the right remediation strategy.

Although the proposed solution uses sensors on several architectural levels, it is targeted mainly at detecting different types of DoS attacks. The detection is based on the information about known attacks stored in the ontology. Detection of unknown anomalies is not possible with this solution. Since a central decision engine is used, scalability is also a problem of the architecture for growing network size or an increasing number of events. Finally, historical data is not taken into account in the detection process, missing another opportunity to eliminate false positives.

Gorton [61] argues that the usage of a diversity of sensors on several architectural levels raises the chance to detect an attack, because the sensors may reinforce each other. However, this requires managing and correlating the higher number of events and alerts. Different solutions have been developed in the area of intrusion correlation, targeted to the reduction of alerts a security officer must address. The potential to detect anomalies using these different information sources, however, is not the focus of these solutions.

6.3 Design

In this section, the design of the four proposals that aim to increase virtual machine security is presented.

6.3.1 Update Checker

The first step towards increased virtual machine security is the identification of outdated or insecure software installed in virtual machines. This is a crucial step in the process of keeping virtual machines up-to-date. While scanning for availability of software updates is easy to perform for running virtual machines, because of the commonly used package management systems on Linux platforms, it is hard for dormant virtual machines. A trivial approach would be to regularly start each virtual machine, scan for availability of software updates, install the updates, if available, and shut the virtual machine down again. While this is technically feasible for individual virtual machines thanks to automation, it does not scale to the numbers of virtual machines in existence nowadays and the required scanning intervals, both in terms of time and required resources.

The *Update Checker*, the first proposal in this chapter, is able to efficiently scan a potentially huge number of Linux-based virtual machines for availability of software updates. To be able to scan running and dormant virtual machines likewise, the Update Checker utilizes a central database as its main source of information. The database stores information about the installed packages as well as configured repositories for each of the registered virtual machines. It enables the Update Checker to execute the scan locally without booting any of the virtual machines at all. With this approach, the Update Checker is currently able to scan both running an dormant virtual machines that use either the *apt/dpkg* or *yum/rpm* as package management system and thus all major Linux distributions[1]. As the result of its scan, the update checker returns either the number of available updates or details about each of the available updates for each registered machine.

The key idea is that the scanning process is executed continuously without requiring any interaction of the user. The results of the scanning process are stored in the database and made available via an API, allowing virtual machine management systems to integrate the results into their interface. It is then up to the user to take the appropriate actions.

This section starts with an introduction of package management system basics and versioning schemes that are important for the functioning of the Update Checker. Afterwards, the architecture of the Update Checker is presented. The section concludes with a usage scenario for the Update Checker in a Virtualized Grid Computing environment.

6.3.1.1 Package Management Systems

Package management systems are the foundation of most Linux systems. Their goal is to blur the line between applications and operating system features, by integrating applications into the tools used to maintain the operating system itself [102]. This does not only cover the installation of applications, but also their removal, configuration, and updating. Especially the latter is of importance for the security of computer systems.

They supersede the concept of specific setup tools for installing and removing individual applications as well as specific update daemons for them. This less user-friendly concept is still used in all but the newest Windows and OS X systems, in that centralized software stores with automatic update facilities have been introduced. Package management systems also solve the dependency problems that sometimes occur with badly written setup tools that do not install all required libraries.

There are few basic concepts of package management systems that are important for the design and implementation of the Update checker:

Package — A container for anything from an application or library through to documentation, configuration files, or even source code. It consists not only of its content, but also of some amount of metadata describing the content. This metadata is the key enabler for package management. The metadata is comprised of a unique name for the package, version information, dependencies and conflicting packages, and more.

[1] Support for other package management systems is possible by implementing a small compatibility layer that enables reading of the respective package databases and repositories.

Package Database — A database used by the package management system to keep track of installed packages and their metadata. This overview of the complete system allows the package management system to easily resolve dependencies, prevent conflicts, and determine outdated software.

Repository — A collection of packages provided for a Linux distribution. It is the primary source of software for the package management system to download new and updated software from. Typically, repositories are hosted on servers reachable via the Internet , but a repository can also be stored on a removable medium like a DVD.

Repository Database — The consolidated metadata of all packages available at a repository. It works like an index in a database, allowing the package management system to efficiently search the repository for specific software without downloading all the individual packages.

Repository Configuration — A set of configuration files on a system that selects the repositories used by the package management system for software to install. It typically contains information about the location of the used repositories in terms of URLs or other descriptions.

When searching for new software or updates for software already installed, the package management system first downloads the repository databases of all enabled repositories. Depending on the task, it either searches for a package name, does a full-text search in the package descriptions, or retrieves the version information of specific packages. It then checks if all dependencies are satisfied and no conflicts are introduced when a new package or update is installed. If necessary, it extends the set of packages to be installed with other required packages.

Version Identifiers

Version identifiers are an important factor for any package management system. They are the foundation of automatic updates, but also play an important role in the definition of dependencies or conflicts. To provide maximum flexibility for the developers of software, there is no strict version number scheme enforced by the package management systems. This leads to a myriad of different schemes being used at the same time (an few examples can be found below).

The version identifier commonly used by package management systems is a combination of the actual version number and two other components: the *epoch* and a build identifier often called *release*. This extension ensures that versions can be reliably compared even when an unusual version number scheme is used. The following version identifier format is used in different package management systems [12, 33]. Table 6.1 contains a description of the fields that constitute the version identifier. The *epoch* is the most significant component in the version identifier, the *release* the least significant.

```
[epoch:]version[-release]
```

Version Identifier Scheme Used by Package Management Systems.

Table 6.1 Fields Constituting the Version Identifier. These fields are used compose the version identifier commonly used by package management systems.

Field	Description
epoch	The *epoch* field is an unsigned integer value that is used in cases where the actual version number scheme cannot be parsed by the package management system or to cope with changes in the scheme. By increasing the *epoch*, a newer version can be recognized, even if the actual version number is lower than that of the previous version. Only a minority of packages use the *epoch* field, and some package management systems even discourage its use entirely [12]. If the *epoch* is 0, it is usually omitted.
version	The *version* field contains the version number of the upstream package, i.e., the source package released by the software author. As already said, there are almost no restrictions concerning the format of the version number. Note that contrary to what one might expect, the version number might contain colons, if an *epoch* is present, and dashes, if a *release* is present. Otherwise, these characters must not be used in the version number, because it would thwart splitting the version identifier into its three components.
release	The *release* field is used when a package is rebuilt, i.e., after a critical patch has been applied. While the value of the *version* field does not change in this case, the *release* field is incremented to reliably recognize the updated version. Like the *epoch* field, this field can be empty and thus omitted. This field was supposed to contain an unsigned integer like the *epoch* field, but there are almost no restrictions to the contents of the *release* field and arbitrary values are used in practice. (The Debian Package Manager calls this field *revision*. The term *release* is used throughout this thesis, independently of the package management system.)

A list of exemplary packages in Ubuntu 14.04 and their version identifiers is shown below to illustrate the variability in version number schemes. The components have been highlighted to simplify their mapping.

```
apparmor                    2.8.95~2430-0ubuntu5.3
autotools-dev               20130810.1
bochs                       2.4.6-6
bison                       2:3.0.2.dfsg-2
flashplugin-installer       11.2.202.521ubuntu0.14.04.1
python3                     3.4.0-0ubuntu2
gcc-4.8                     4.8.4-2ubuntu1~14.04
fonts-opensymbol            2:102.6+LibO5.0.1~rc2-0ubuntu1~trusty1
```

Exemplary Version Numbers for Various Packages in Ubuntu 14.04.

6.3.1.2 Architecture

The primary goal of the Update Checker is detecting obsolete software in (dormant) virtual machines, thus the chapter focuses on virtual machines. Nevertheless, the Update Checker can be used for physical machines as well.

The concept of the Update Checker is to build a central database that contains all the information required for the task of checking for updates. This includes the list of installed packages including the corresponding version identifiers as well as the list of repositories that are enabled for each virtual machine. This information has to be imported into the central database when the virtual machine is first registered, and updated after each change of the virtual machine, i.e., after new software has been installed or software in the virtual machine has been updated.

Since the Update Checker is not targeted at a single Linux distribution (compared to, e.g., Landscape that supports only Ubuntu [24]), at least the two prevalent package management systems are supported: apt/dpkg, used for example by Debian and Ubuntu, as well as yum/rpm, used for example by Red Hat and Fedora. Both systems use a specific package database format as well as a specific repository format. While apt/dpkg uses the same plain text file format both for the package database and for the repository database, yum/rpm uses a Berkeley database for the package database and an XML file for the repository database. Nevertheless, this has no influence on the structure of the database used to store the required information, since both systems have the concept of distinct package names and a consistent version identifier scheme in common.

The design of the Update Checker is shown in Figure 6.2. There are specific importers for the package databases and for the repository databases of the different package management systems. This makes the Update Checker easily adaptable to other package management systems. Information about the installed packages of a virtual machine is stored in the *Package DB*. Metadata about the virtual machine, i.e., the time stamp of the import, the enabled repositories used, etc., is stored in the *Metadata DB*. Information about the packages available in the different repositories is stored in the *Repository Cache*. When invoked, the Update Checker takes the information from the databases and the *Repository Cache* and matches installed and available packages to detect obsolete software. The findings are stored in the *Result Cache*.

When a query for the state of one or more virtual machines is issued, the Update Checker first examines the Result Cache to see if the result of that query is already available and returns the cached result unless it is obsolete. Cached results are considered obsolete after a configurable amount of time, depending on factors such as the frequency of updates or the need for security.

If current results are not available in the Result Cache, the Update Checker determines if the package lists of all repositories enabled in the virtual machine are available in the Repository Cache in a current version, i.e., the configured validity period for the package lists has not yet expired. Missing or obsolete package lists are downloaded from the corresponding repository, parsed and stored in the Repository Cache for future use. Finally, the actual scan of the virtual machine is started, comparing

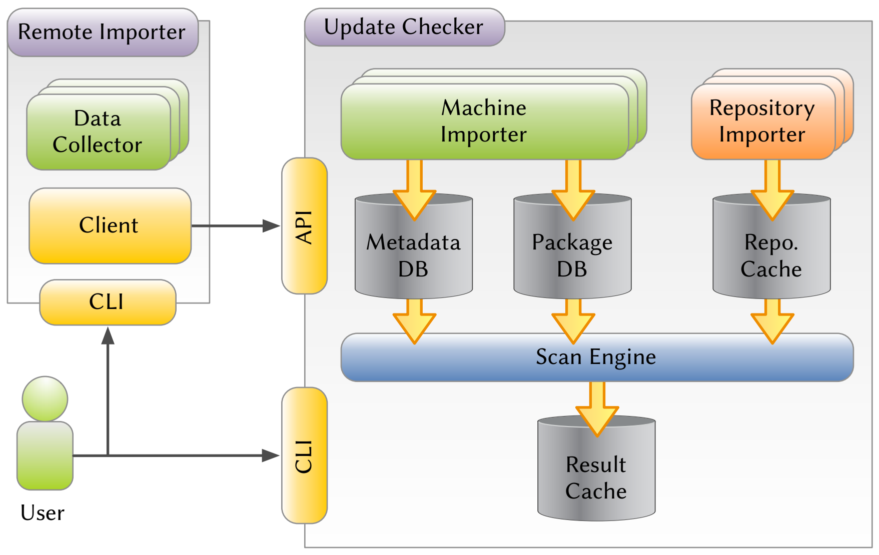


Figure 6.2 Architecture of the Update Checker. The central service of the Update Checker consists package management systems specific importers, the central package and metadata databases, caches for repository contents and results, and the actual scan engine. Furthermore, a Remote Importer exists that consists of package management system specific Data Collectors as well as a client that uploads the collected data to the central service.

the version of each installed package with the versions available in the repositories. Information about outdated packages is then stored in the Result Cache, so that subsequent queries regarding the same virtual machine can be answered faster.

By using the Repository Cache instead of the actual repositories, there is the risk that the Update Checker fails to identify an outdated package. Nevertheless, the Repository Cache is very useful for checking many virtual machines, and by using a small validity period the risk can be minimized.

To help the user to judge whether the identified outdated software poses a risk to the virtual machine, the Update Checker infers information about the priority of an update. Unfortunately, there is no common way to do this for multiple distributions. As a first approach, the source repository of the updated packages is evaluated, since distributions like Debian or Ubuntu use special repositories for security updates. The source of an update can therefore be used as a hint of its priority.

The Update Checker supports queries for the number of available updates for a single or multiple virtual machines as well as queries for details about the outdated packages and available updates for a single virtual machine. The first kind of queries can be used to get a good estimation of the state of the virtual machine: zero available updates means that the virtual machine is up-to-date, whereas one or more available updates means that the virtual machine contains outdated software. If priority information is available, individual numbers for each priority as well as the sum of the numbers are returned. This can either be used in situations where an overview over a number of virtual machines is required, e.g., a list of virtual machines in a management tool like the ICS, or as a status check for a specific virtual machine, e.g., before it is started by the XGE.

The availability of updates itself is sufficient to reason about the threat resulting from the outdated packages, even when priority information is available. The second kind of queries returns a list of outdated packages that can be used to evaluate the status of a virtual machine in detail. The owner of the virtual machine or an administrator can do a threat analysis based on the outdated packages and decide whether immediate action is required or not.

Two different interfaces to the Update Checker are provided: a command line interface (CLI) and an XML-RPC [169] API. The former can be used when an administrator manually wants to execute an update check or register a virtual machine. The latter is designed to facilitate easy access to the status information for other tools (see also the use case below).

The Update Checker can be configured to execute scans at regular intervals, e.g., daily or weekly. This speeds up queries, because the queried information is already available in the Result Cache. These automatic checks also include a notification feature. Virtual machine owners can be informed about obsolete software in their virtual machines via email. Additionally, administrators can be informed about all virtual machines using obsolete software to get an overview of the security all virtual machines running on their infrastructure.

To ease the registration of virtual machines, the Remote Importer is provided. It uses package management systems specific Data Collectors to gather the information required for the Update Checker, sends it to the machine the Update Checker is running on and triggers the registration process. It might seem cumbersome to manually re-register virtual machines after every change, but with the Remote Importer it is merely a single command. Furthermore, it can be easily automated when software for management and maintenance of virtual machines is used.

6.3.1.3 Virtualized Grid Computing Use Case

A potential use case of the Update Checker is shown in Figure 6.3. This is a setup used in Virtualized Grid Computing environments. The ICS is used to create and maintain virtual machines, and to prepare them for use in the Grid. The XGE is a Scheduler for Virtualized Grid Computing that schedules jobs for execution in virtual machines. When a job is scheduled or about to be executed – depending on the need of security – the XGE retrieves information about the update status of the virtual machine selected for the execution of the job. Depending on the status, the XGE can reject the job or prevent the virtual machine from being started, respectively.

6.3.2 Centralized Update Process

The second step towards increased virtual machine security is to install available software updates to all virtual machines, after the Update Checker presented above has identified them. Again, the trivial approach would require starting each virtual

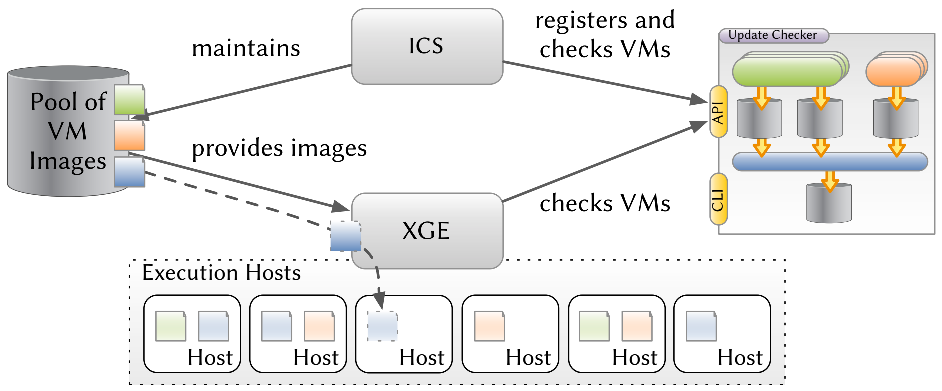


Figure 6.3 Virtualized Grid Computing Use Case. This figure shows the usage of the Update Checker in a Virtualized Grid Computing Environment. The XGE uses the Update Checker to determine if a virtual machine is up-to-date before it is started.

machine, installing the software updates, and shutting the virtual machine down again. Although scanning for updates can be combined with installing available updates to reduce the number of virtual machine start-stop cycles required for maintenance, but the fundamental scalability problem of this approach cannot be solved in this way.

The second proposal presented in this chapter is a concept that solves the scalability problem of update installation in a large number of virtual machines by deploying the Virtual Machine Image Composition Technique presented in Chapter 4. Empowering administrators to fix vulnerabilities in a large number of virtual machines by solely installing updates in a small number of shared layers – sometimes even a single shared layer – is the fundamental idea of this approach. When used in this way, shared layers offer not only a method to reduce deployment times, but also a way to centrally update a large number of virtual machines. Any layered virtual machine that is built on one of those centrally updated shared layers will be protected from the corresponding vulnerabilities as soon as it is rebooted and its composite disk image is rebuilt using the updated version of the shared layer.

The concept is accompanied by a set of tools that support the centralized update process by taking care of the remaining tasks in the centralized update process: ensuring that the installed updates are effective in the layered virtual machine and providing an updated package database that reflects the installation of the updates to the guest operating system inside the virtual machine. Both steps are required because of the semantics of the underlying composition technique, where changes in higher layers take precedence over changes in lower layers (see Section 4.3.2.2). Additionally, a tool is provided that supports efficient distribution of updated base layers.

6.3.2.1 Updating Software

Virtual machine composition allows efficient updating of virtual machines, because the update only needs to be applied to the shared base layer. As a consequence, all layered virtual machines built upon this base layer are automatically updated. This approach works, because all changes in the base layer are visible in the composite disk image, unless they are hidden by corresponding files in the user layer.

A drawback of this approach is that it leads to an inconsistent package database in the user layer and thus in the resulting composite disk image if the package database has ever been modified in the composite disk image. Unfortunately, such modifications of the package database are the rule, because the base layer only contains a set of commonly required packages and thus the users of layered virtual machines typically install additional software for their specific needs using the package management system[2].

Figure 6.4 illustrates this problem. When the user installs software in his or her layer, the union mount used for virtual machine composition copies the package database from the base layer into the user layer before any changes are made to it (Figure 6.4b). Thus, any changes in the package database of the base layer, e.g., updating a package, are not visible in the composite disk image anymore: the package database is inconsistent (Figure 6.4c).

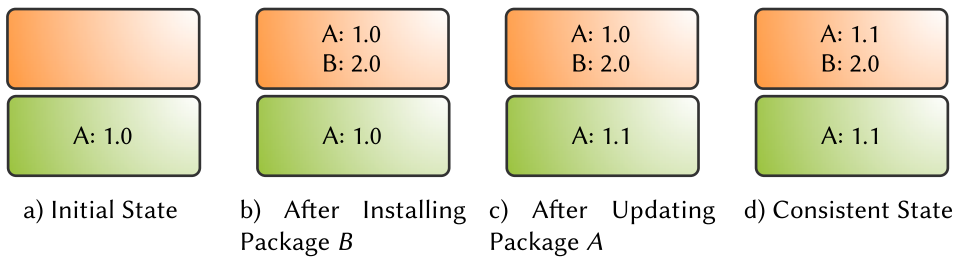


Figure 6.4 Package Databases in a Dual-layered Virtual Machine. For each layer, the contents of a hypothetical package database are shown. Note that in a composite disk image only the upper package database is visible. A and B depict packages with their corresponding version number. The databases are shown in different states: a) after the creation of the base layer (consistent), b) after installing package B in the user layer (consistent), c) after package A was updated in the base layer (inconsistent), and d) after package A was updated in the base layer (consistent).

A more serious problem can arise, because the package management system is not aware of the software installed in the user layer while the updates are installed in the base layer. In the worst case, the updated base layer is incompatible to the user layer. There are two kinds of incompatibilities that can occur: *masking incompatibilities* and *relation incompatibilities*. Although very rare if the centralized update process is used deliberately, the possibility of such incompatibilities needs to be kept in mind.

Masking incompatibilities can occur in two forms. First, a package updated or added in the base layer is hidden by another version of the package installed in the user layer or whiteout files that have been created by deleting the package in the user layer. Second, a package removed in the base layer can still be available if it is also installed in the user layer. In both cases, the change in the base layer is not effective in the composite disk image.

[2] Installing a new package is not the only operation that leads to inconsistencies in the package database. The other operations are updating or removing an installed package.

Relation incompatibilities can also occur in two forms. First, the update in the base layer can breaks dependencies of packages installed in the user layer either by removing a package or updating a package to an incompatible version[3]. Second, the update in the base layer can constitute conflicts by adding a package that is in conflict with a package installed in the user layer.

In the following sections, two tools are described that deal with those problem. The first tool merges the package databases in the updated base and the user layer and creates a consistent version of the package database (Figure 6.4d). Additionally, this tool detects masking incompatibilities. The second tool is used to detect relation incompatibilities.

6.3.2.2 Merging the Package Database

Merging the package databases in the updated base and the user layer is a problem related to the three-way merge that is an important foundation of revision control systems. The general three-way merge for files is outlined in Figure 6.5a: Two files *A* and *B* with a common ancestor *C* should be merged to yield the resulting file *D* that contains the changes from both *A* and *B* relative to *C*. The three-way merge algorithm is implemented in the `diff3` [78] utility ubiquitously available on Unix systems.

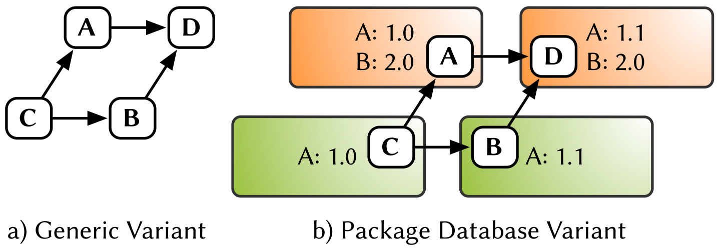


Figure 6.5 Three-way Merge. This figure shows the concept of a three-way merge both for regular text files – the original domain of the algorithm – and for package databases.

Figure 6.5b shows the application of the three-way merge to the package databases shown in Figure 6.4. Unfortunately, the three-way merge algorithm cannot be borrowed in its original form, because the union mount semantics. Since files in the user layer hide files in the base layer, entries in the user layer's version of the package database have to hide corresponding entries in the base layer's version of the package database. Furthermore, the merge algorithm cannot blindly merge entries, but has to observe the state of the package[4].

The merge algorithm works by determining all packages that are updated, added or removed in the updated base layer relative to the original base layer by comparing corresponding entries in the two package databases. Then, it copies all entries that fall into none of the three categories from the user layer's package database to the resulting package database. For each of the entries in the three categories, the algorithm does a

[3] It is possible to define dependencies with version restrictions.
[4] There are around half a dozen states besides *installed* in Debian's dpkg, e.g., unpacked and config-files.

series of checks to determine which entry has to be copied to the resulting package database: the entry from the user layer's or the updated base layer's package database. An additional function of the checks is the detection of masking incompatibilities. A detailed description of the algorithm can be found in the implementation section.

6.3.2.3 Detecting Relation Incompatibilities

After the package databases of the old base, base, and user layers have been merged successfully, the relation incompatibility detection algorithm can be applied to the merged packaged database. The algorithm iterates over every package in the package database and checks whether its dependencies are satisfied, i.e., all packages it depends on are installed, but no conflicting packages are installed. An improved version of this algorithm is presented in the implementation.

6.3.2.4 Efficient Distribution of Updated Base Layers

A problem related to updating a shared base layer is the efficient distribution of the updated base layer to the execution hosts that have copies of the base layer's older version in their image cache. Obviously, the updated base layer could just be copied to the execution host again, but for small updates this is not efficient. If the base layer is stored in the Image Store, the process to update exported image files (see Section 5.4.6.2) can be used.

Another approach is to borrow the concept of an *update layer* that is used to improve the MIS reimport times after an update (see Section 5.4.6.1). The update is installed in the update layer that is then copied to the execution hosts. There, the update layer is *applied* onto the older version of the base layer. This approach is depicted in Figure 6.6. Applying in this case means the following: First, find all *whiteout files* in the update layer and delete the corresponding files or directories in the base layer. Second, find all *opaque directories*[5] in the update layer, recursively delete the corresponding directories in the base layer, and create empty directories as replacement. Finally, copy all files contained in the update layer to the base layer, whereby existing files in the base layer are replaced with the files from the update layer.

6.3.2.5 Limitations of the Approach

The centralized update process can solve the scalability problem, but introduces new compatibility problems. In case of vital security updates, however, the benefits of installing these updates may outweigh the risk of creating incompatibilities with individual user layers. An example of such a vital update is the fix of the famous OpenSSH Bug in Debian [32] that reduces the possible number of generated SSH keys to 65535. This bug enables an attacker to gain root access to an affected (virtual) machine typically in less than 20 minutes using precalculated keys [165]. By utilizing the cumulative computing power of virtual machines available in the Cloud or at a

[5] The meaning of whiteout files and opaque directories is described in Section 5.3.4.1.

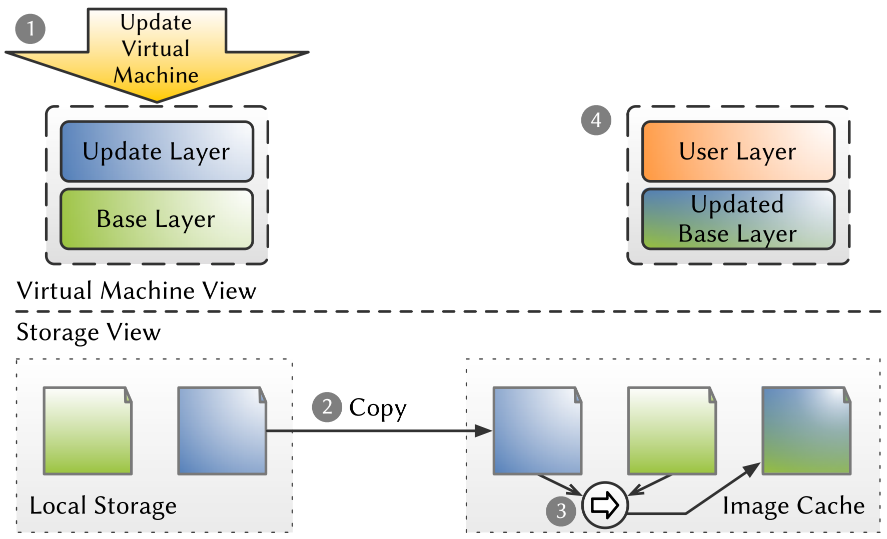


Figure 6.6 Efficient Distribution of Updated Base Layers. The figure shows the usage of an update layer to efficiently distribute update: 1) the base layer is updated using an update layer, 2) the update layer is copied to one or more execution hosts, 3) the update layer is applied to the base layer in the image cache of the execution host, and 4) any virtual machine started afterwards uses the updated base layer. Applying the update layer (Step 3) typically overwrites the old version, because it is applied in place.

Grid site, the time and cost of using this exploit at a larger scale may be significantly reduced. This bug posed a threat to the entire Internet by enabling attackers to leverage "free" resources in Virtualized Grid and Cloud Computing environments to attack external sites.

It is important to use the centralized update process only for Linux distributions using a non-rolling release model, because updates in a rolling release model can introduce new functionality and thus have a higher risk of creating (potentially undetected) relation incompatibilities. Fortunately, most Linux distributions use a non-rolling release model, especially distributions used for servers and thus in Virtualized Grid and Cloud Computing environments.

Another problem besides incompatibilities introduced by the update are updates that require changes to configuration files. These files have a higher probability of being changed in the user layer compared to binary files (packages), because changes to configuration files might be necessary to adapt a virtual machine to the needs of its owner. As of writing, no automatic solution for this problem is provided. It is partly solvable by executing the script included in the update that implements the change in the configuration file when a virtual machine is started with the updated base layer for the first time. But this only works if the change can be cleanly applied, because it is not possible to ask the user what to do at the time of executing the script.

6.3.3 Online Penetration Suite

The last two proposals facilitate regular checks for the availability of software updates and timely installation of identified software updates. However, these proposals are not sufficient to secure virtual infrastructures. Even if the operating system and software installed in all virtual machines are kept up-to-date, some potential vulnerabilities remain:

- Misconfiguration of firewalls
- Presence of insecure or unnecessary services
- Misconfiguration of services
- Known vulnerabilities in services with no available updates
- Improperly secured shares

Virtual machines used in Cloud Computing environments are subject to external attacks, either attacks targeted directly at a specific virtual machine or attacks by worms or scripts that randomly select their targets. To further increase virtual machine security, virtual machines have to be analyzed continuously. If vulnerabilities are found, proactive countermeasures need to be taken if possible, but at least the owners of the virtual machines need to be notified, so they can take appropriate actions.

A system called *Online Penetration Suite* (OPS), the third proposal in this chapter, performs online-checking of virtual machines for the kind of vulnerabilities described above. The OPS is a generic vulnerability-scanning framework for virtual machines that utilizes one or more vulnerability scanners as scan engines. Since the vulnerability databases of different scanners likely deviate regarding contained vulnerabilities, a combination of multiple scanners presumably leads to an increased vulnerability detection rate. The OPS generates a combined report that does not only contain the reports of all deployed scan engines, but also a summary of all findings, providing a quick overview of the vulnerabilities of a virtual machine. To facilitate further processing of the results, the reports are provided in a machine-readable format if required. Dormant virtual machines are automatically started before a scan and shut down afterwards by the framework using a hypervisor-independent library. This allows automatic testing of virtual machines to detect known security vulnerabilities. A use case for automatic scanning of virtual machines is presented in Section 6.3.3.2.

Scans can be done in two different phases of a virtual machine's lifecycle: in the storage phase and in the deployment phase. In the storage phase, the scans can be done either periodically or during periods with low load as part of the continuous maintenance. An alternative is to execute the scan during the deployment phase immediately before a virtual machine goes live, at the cost of prolonging the deployment time. A trade-off between fast deployment and increased security – because of potentially newer vulnerability databases used for the scan process – is necessary, to decide which approach is better suited for an individual use case. Generally, the former approach is better suited for Cloud environments, where the user can start his virtual machines interactively and minimal deployment times are required to provide instant scalability for applications. In Virtualized Grid Computing, on the other hand, the latter approach can be used, because of the characteristic non-interactive batch processing model. Additionally, the batch scheduling system is able to forecast the deployment time of a virtual machine and can thus schedule its scan with the OPS shortly before deployment time, combining the advantages of both approaches.

6.3.3.1 Architecture

The OPS is divided into two parts: the front end that controls the scan process and generates the reports and the back end that that orchestrates both the virtual machines to scan as well as the vulnerability scanners. The architecture of the OPS is shown in Figure 6.7 with two exemplary adapters for *OpenVAS* [112] and *Nessus* [154] vulnerability scanners.

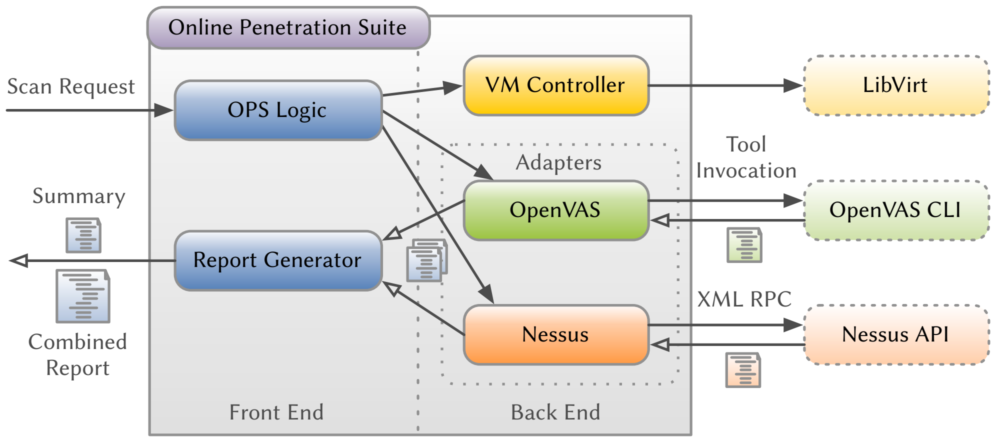


Figure 6.7 Architecture of the Online Penetration Suite. The OPS is divided into a generic front end and a back end with adapters for different vulnerability scanners and utility libraries. The figure depicts the control flow within the OPS and the reports in the OpenVAS (green), Nessus (orange), and OPS (blue) formats.

The *OPS Logic* module controls the scan processes. It configures the security scanners selected by the user, boots the virtual machines to scan (if required) and starts the actual scan processes. Since the vulnerability scanners are third-party products with distinct characteristics and modes of operation, they are abstracted by adapters that hide the differences and provide a unified interface to start and monitor the vulnerability scanners. They enable the OPS not only to start the actual scans, but also to watch the scanners during the execution to detect any failures and react accordingly.

To scan virtual machines for vulnerabilities, the OPS needs two input parameters: the IP addresses or unique identifiers of the virtual machines to scan depending on whether they have already been started or not, respectively, and the names of one or more vulnerability scanners to use for the scan. If no scanners are provided, the OPS enables all scanners by default. Dormant virtual machines are deployed to a designated scanning host from the Marvin Image Store and started before scanning.

The *Report Generator* module collects the reports from the enabled scanners and generates the reports: a summary that contains the number of detected vulnerabilities categorized by a risk factor and a combined report that contains the results from the individual security scanners in a unified format. To be able to process the reports, the Report Generator expects the reports of the vulnerability scanners to be in the unified OPS format. Thus, the adapters have to convert the reports from the native format of the scanner before returning them to the Report Generator.

The back end of the OPS contains the *VM Controller* that can start and stop virtual machines on the dedicated scanning host using the libvirt [21] library as well as the vulnerability scanner adapters. Currently, the OPS supports two scanners: OpenVAS [112] and Nessus [154], both well-known and established security-products.

The clear separation between front end and back end created by the use of adapters and the unified OPS format makes the OPS easily extensible with regard to new scanners. No vulnerability scanner dependent code is used in the front end, therefore such an extension solely needs to focus on the implementation of an adapter for the new vulnerability scanner.

6.3.3.2 Virtualized Grid Computing Use Case

The Virtualized Grid Computing use case for the Update Checker presented in Section 6.3.1.3 can also be applied to the OPS, because the Update Checker and the Online Penetration Suite have the same task – to evaluate the security of a virtual machine –, although they use other means to do their job. In fact, they complement each other and they can be combined in this use case. The only difference is the amount of time required: scanning for outdated software using the Update Checker is a much faster process than scanning for vulnerabilities using the OPS. Consequently, it is more sensible to trigger OPS scans as regular maintenance operations using the ICS than to scan a virtual machine before it is started by the XGE.

6.3.4 Security Monitoring

To detect attacks in Virtualized Grid and Cloud Computing environments, it is useful to be aware of as many anomalies in the behavior of the system as possible. Theoretically, every event that occurs in one of the different layers of a virtualized system can be an indicator for an anomaly, e.g., established network connections, creation or termination of processes or even user or process activities beyond regular working hours.

The sensors of a monitored environment cannot make the decision about what is a normal or unknown system behavior. Instead, a Complex Event Processing (CEP) engine is responsible for processing all the information sent by the sensors and deciding what can be viewed as normal system behavior based on that information. Through dynamic deployment of further sensors, it is possible to eliminate false positives and verify findings.

Therefore, the architecture of the anomaly management system consists of a secure and trusted virtual machine called *ACCEPT-VM* and a set of sensors and actors. The

ACCEPT-VM contains the main analysis components of the system: the CEP engine, the *Action Framework*, as well as the *Event Store*. Sensors and actors are deployed on every layer of each virtual machine: in the hypervisor or in the guest operating system and in the application layer of every virtual machine. All of these sensors continuously deliver a stream of information to the ACCEPT-VM, whereby each actor is able to execute a specific set of actions on its corresponding layer in order to respond to any detected problem. The overall architecture of the monitoring system is shown in Figure 6.8.

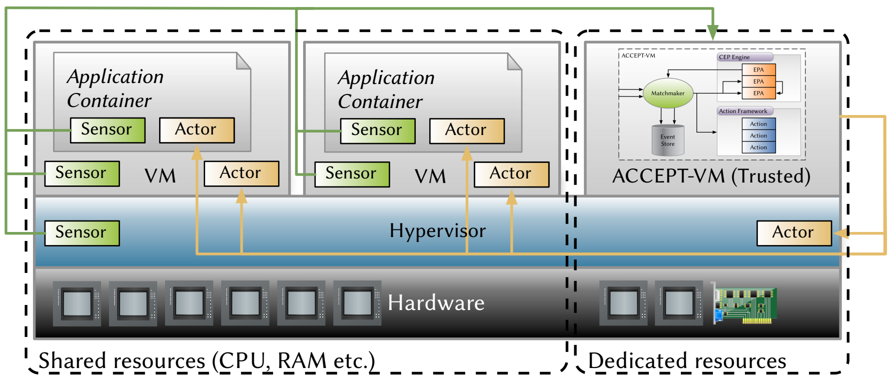


Figure 6.8 Overall Architecture of the Monitoring System. The figure shows all components of the monitoring system: sensors and actors in all layers of the monitored environment and the ACCEPT-VM that contains the CEP engine, Action Framework and Event Store.

6.3.4.1 Sensor Framework

As stated above, sensors are deployed on all layers of the virtualized system: in the hypervisor, in either the user or kernel space of the guest operating system and in application containers. On the one hand, this allows the monitoring system to gather as much information as possible about anything that happens in the system. On the other hand, this facilitates comparison and correlation of the information from different layers. The latter is of special importance to detect malicious software that has rootkit-like capabilities, e.g., hidden network connections that are not visible in the user space of the guest operating system, but in the hypervisor. All sensors should be configurable, e.g., in terms of amount of collected data or measurement intervals, and minimally invasive with regard to the performance of the monitored system and resource consumption.

There are several possibly interesting metrics to be gathered on each layer. Some of the expected events to be gathered can be found in the following list:

Hypervisor Layer — Network traffic, system calls from within virtual machines, process lifecycle information.

Operating System Layer — File access, network sockets, resource utilization.

Application Layer — Java Virtual Machine (JVM) information (heap utilization, thread count, library calls).

Sensors can monitor traditional characteristics of resource usage, e.g., percentage of processor usage or memory consumption, as well as metadata about all running processes in a system, e.g., system calls, network traffic, and read/write memory access. This low-level information greatly improves opportunities for detection of more sophisticated attacks, e.g., malicious polymorphic code, hidden processes, or ongoing memory corruption exploits. On the hypervisor layer, virtual machine introspection is used for acquiring monitoring data; on the operating system and application container layer, the sensors are running as privileged user processes.

Sensors installed on the application layer are used to monitor application behavior. This could be an application container such as Tomcat or JBoss or a bare Java Virtual Machine. A number of metrics is gathered by these sensors, e.g., changes of the memory heap, number of threads, number of loaded Java classes, or statistics of the garbage collector.

6.3.4.2 Action Framework

Like sensors, actors are deployed on all layers of the virtualized system to facilitate triggering appropriate countermeasures for any detected anomaly. Examples of actions that can be triggered using actors of the appropriate layer can be found in the following list:

Hypervisor Layer — Start, stop or pause a virtual machine. Block or shutdown network interfaces.

Operating System Layer — Start, stop, terminate processes or network connections. Delete users or files.

Application Layer — Launch the garbage collector, solve deadlocks. Relaunch or terminate the application container or even remove components from the latter.

More complex actions include a migration of a compromised virtual machine from the productive network to a separate honeypot network in order to detect possible malware. To facilitate the execution of complex actions and enable administrators to implement countermeasures for arbitrary incidents, the actors need to be as flexible as possible. This is realized by specifying actions using a scripting language. Obviously, this opens a door for abuse of the Action Framework for malicious purposes. To mitigate this risk, all actions have to be cryptographically signed.

6.3.4.3 ACCEPT-VM

The ACCEPT-VM is the central instance of the monitoring system and thus responsible for event processing and triggering of actions. It consists of the CEP engine, the Action Framework, an Event Store and the Matchmaker (see Figure 6.9). Because of its central role, it is also the main target of attackers. Consequently, it has to be hardened against attacks to prevent attackers from evading detection by compromising the ACCEPT-VM.

The hardening consists of several measures, e.g., reducing the number of services to the minimum, implementing Mandatory Access Control and enable integrity checks on the file system. Because of this hardening, the ACCEPT-VM is considered a trusted virtual machine contrary to all other virtual machines in the system.

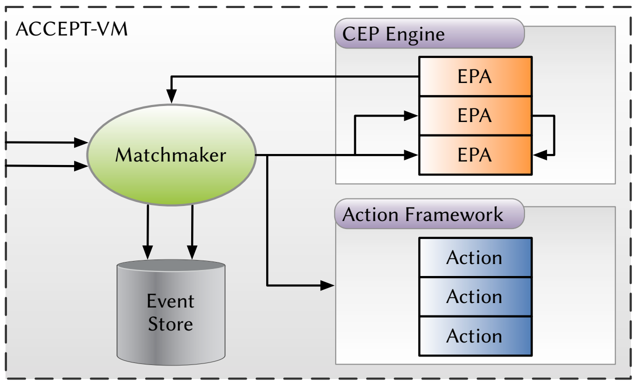


Figure 6.9 Architecture of the ACCEPT-VM. This figure shows the central components of the monitoring system: the CEP engine that analyzes the stream of events from the sensors, the Action Framework that executes actions triggered by the CEP engine on various layers of the virtualized system, the Event Store that stores events for later offline analysis, and the Matchmaker that Architecture of the ACCEPT-VM. (Source: [16])

The capability of performing analysis on a large amount of data on the input stream coming from the sensors is ensured by two main approaches: First, the Event Processing Agents (EPA) running in the CEP engine are assisted by pattern matching techniques. Second, the inherent control- and data-parallelism of EPAs are used to increase the number of events that can be processed.

Incoming events have to be stored for a certain period of time to facilitate offline analysis of the data gathered from the sensors. This is an important addition to the real-time analysis of events that is conducted in the CEP engine. The Event Store is a storage system that is able to cope with large amounts of stream data while providing efficient access to the stored data.

The Matchmaker is a mediator between sensors, EPAs, the Event Store and the Action Framework. It routes incoming events to all EPAs that are interested in the corresponding event type and to the Event Store that stores events for offline analysis. Actions triggered by those EPAs are routed to the Action Framework that takes care of performing the selected actions using the actors deployed in the virtualized system.

The CEP engine performs a real-time analysis on the events using a set of EPAs that monitor the incoming sensor data using continuous queries. These queries are specified in a simple query language that enables combining information of multiple event streams that correspond multiple sensors – potentially on different layers – or even the output of other EPAs.

6.3.4.4 Mode of Operation and Example Scenario

The general functioning of the monitoring system is depicted in Figure 6.10. It is characterized by the sense-detect-react-cycle depicted by the blue loop. In the first step of the loop, the various sensors gather data and send corresponding events to the CEP engine. In the second step of the loop, the active EPAs monitor the incoming data and trigger one or more actions via the Action Framework if there specific conditions are met. In the last step of the loop, the actors situated on various layers of the virtualized system execute triggered actions. This loop repeats perpetually. Once an intruder starts an attack on the system, traces of this attack will be delivered to the CEP engine in the event data streams, and corresponding EPAs will trigger appropriate actions that implement countermeasures against the attack. To illustrate this approach in action, an exemplary anomaly detectable via double-entry accounting of the hypervisor and operating system layers' port lists is shown below.

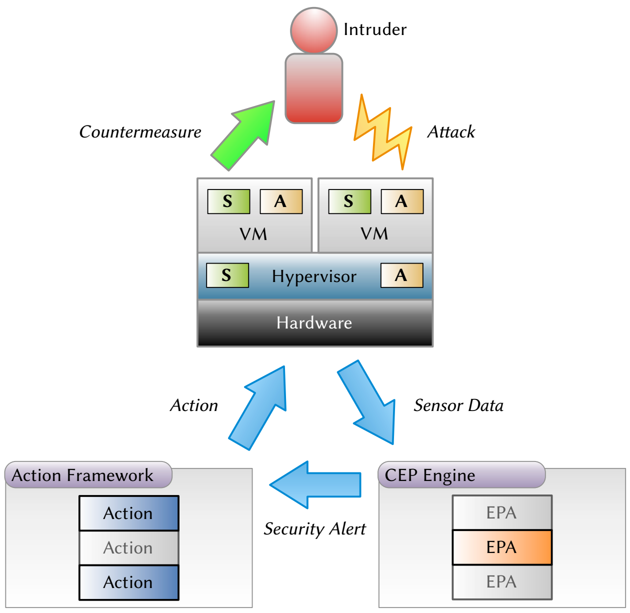


Figure 6.10 The Sense-Detect-React-Cycle. This figure shows the functioning of the monitoring system. The attack initiated by the intruder is detect by an EPA in the CEP engine (marked orange) that triggers two actions via the Action Framework (marked dark blue), which in turn implement a countermeasure against the attack in the affected virtual machine(s).

In the example scenario, an attacker has successfully installed a backdoor in a monitored virtual machine. Its presence is hidden using a rootkit technology, i.e., a modification of the operating system and its user space interfaces. Even though the backdoor is listening on an arbitrary TCP port, the operating system's user space tools will not list its process and the listening socket.

Sensors — To detect the backdoor, at least two different sensors are required. The first sensor is running within the virtual machine and utilizing standard tools such as `netstat` to check for any listening sockets. Since the backdoor is well hidden, this sensor will not report the security breach.

The other sensor is inspecting the network state of the virtual machine on the hypervisor layer. Since this sensor is running outside of virtual machine and thus the modified guest operating system, it is not affected by the backdoor's stealth features. On this layer, an event is generated for the detection of a newly opened port in the virtual machine.

Analysis — A newly opened port is the first sign of a potential anomaly. Furthermore, by comparing data from both sensors that deliver information about listening sockets from inside and outside of the virtual machine, it can be concluded that this really is a security related anomaly. A regular service installed in the virtual machine would not be hidden within the virtual machine. Especially the conflicting sensor information is a clear sign of an attack.

Action — As a result of this attack, actions should be taken to eliminate the threat as much as possible. One such action could be to block all communication from and to the backdoor's port on the hypervisor layer. This prevents the attacker from extracting information or further using the infected machine. Another step that should be taken is to isolate and possibly terminate the processes involved in the infection. For forensics purposes, taking a snapshot of the virtual machine and generating a dump are suitable next steps after the immediate danger is averted.

6.4 Implementation

In this section, the design of the four proposals that aim to increase virtual machine security is presented.

6.4.1 Update Checker

This section describes the implementation of the Update Checker, working from the top to the bottom of Figure 6.2. First, the Machine and Repository Importers and their sources of information are described for both the Debian Package Manager (dpkg) and Advanced Packaging Tool (apt) used by Debian, Ubuntu, and its derivatives and the RPM Package Manager (rpm) and Yellowdog Updater, Modified (yum) used by RedHat and Fedora as well as related distributions like CentOS and Scientific Linux, respectively. Afterwards, the internal databases and caches and the Scan Engine are described. This section is concluded with details about the Remote Importer.

The Update Checker is implemented in the Ruby programming language (Version 2.1 or greater). For parsing the various databases, the *Nokogiri* and *IniParse* libraries are used. Version comparison is done using the *dpkg-ruby* library.

6.4.1.1 Machine Importer

The Machine Importer is responsible for importing the list of installed packages and their versions from the package database of a virtual machine into the Package DB. Furthermore, it imports the list of enabled repositories from the repository configuration of a virtual machine into the Metadata DB. This step includes parsing the information retrieved from the package management systems and removing extraneous information before the data is stored in the internal databases of the Update Checker (see the examples below). The formats used in the internal databases are described in Section 6.4.1.3.

Each of the considered package management systems uses a database in a distinct format to keep track of the versions of installed packages, dependencies between packages, files belonging to each package, etc. Additionally, they use one or more configuration files – again in a distinct format – to store information about the repositories that are enabled for the machine. A machine importer tailored to the specific format of the package management system is used to gather the required information and store it in the Package DB and Metadata DB, respectively. In the following, the format of the databases and configuration files are described and corresponding code to parse the information is presented.

Package Databases

Debian Package Manager – `dpkg` The package database of `dpkg` is stored in `/var/lib/dpkg` and consists of several plain text files. For the Update Checker, only a single file named `status` is relevant, which contains the metadata for each package installed on the system. The remaining files are used for other features of `dpkg` such as *alternatives*, a way to set a default package to use in case multiple installed packages provide alternative versions of a given software[6], and are thus not processed by the Update Checker.

The `status` file contains an entry for each installed package, and each entry consists of around twenty key-value-pairs (the actual number of pairs may vary between packages). Each key-value-pair may span a single or multiple lines. In the latter case, all lines starting with the second have to be indented by a single space character to be easily identifiable as continuation. A single blank line separates the entries of two packages.

Only four of the key-value-pairs are relevant for the Update Checker: *Package*, which contains the package name, *Architecture*, which contains the target architecture of the binary files contained in the package or `all` for architecture-independent packages, i.e., packages containing no binary files or platform-independent software, *Version*, which contains the exact version of the package, and *Status*, which contains the installation status of the package. The latter is important because `dpkg` keeps the

[6] For example, Debian and its derivatives typically provide multiple versions of the Vim editor with different sets of features.

entries of removed packages in the database if their configuration files were kept, leading to the `deinstall ok config-files` status.

The following snippet shows an excerpt from the dpkg package management database to illustrate the format. The relevant key-value-pairs of the *openssh-server* package have been highlighted.

```
...
Homepage: http://savannah.nongnu.org/projects/acl/

Package: openssh-server
Status: install ok installed
Priority: optional
Section: net
Installed-Size: 709
Maintainer: Debian OpenSSH Maintainers <debian-ssh@lists.debian.org>
Architecture: amd64
Multi-Arch: foreign
Source: openssh
Version: 1:6.0p1-4+deb7u2
Replaces: openssh-client (<< 1:3.8.1p1-11), ssh, ssh-krb5
Provides: ssh-server
Depends: libc6 (>= 2.8), libcomerr2 (>= 1.01), libgssapi-krb5-2(...)
Recommends: xauth, ncurses-term, openssh-blacklist, openssh-blacklist(...)
Suggests: ssh-askpass, rssh, molly-guard, ufw, monkeysphere
Conflicts: rsh-client (<< 0.16.1-1), sftp, ssh (<< 1:3.8.1p1-9),(...)
Conffiles:
 /etc/pam.d/sshd ee93e13ec6aa3f3120c6939a2880a5b6
 (...)
Description: secure shell (SSH) server, for secure access from remote(...)
 This is the portable version of OpenSSH, a free implementation of
 (...)
Homepage: http://www.openssh.org

Package: libpolkit-backend-1-0
...
```

Exemplary excerpt from `/var/lib/dpkg/status` showing the `openssh-server` package.

The methods shown in Listing 6.1 are responsible for parsing the package database of dpkg. The `parse_database_file` method is a wrapper that is expecting a path to a package database file. It reads the contents of the database into a string and passes the string to the `parse_database_content` method that is responsible for the actual parsing. This method is described below. Finally, the `parse_database` method passes the path of the package database file relative to a given root path to the `parse_database_file` method.

The parsing code uses the `split` method to create a list of entries from the database and calls the `reduce` method on that list (Line 21). The block passed to reduce converts each entry into a dictionary with the minimum information for each package (Lines 22 to 26). The conversion in Line 26 requires a list of lists, where each inner list contains exactly two elements: a key and a value. This list is created in Lines 23 to 25: First, each entry is split into a list of key-value strings that are basically the individual lines of the entry as shown in the excerpt above (Line 23). In this step, it is important to

```
17  SYMS = [:package, :status, :architecture, :version]
 ⋮
20  def parse_database_content(content)
21    content.split(/\n\n/).reduce([]) do |list, entry|
22      package = entry
23        .split(/\n(\S+:.*)\n?/)
24        .map { |s| k, v = s.split(': '); [k.downcase.to_sym, v] }
25        .select { |k,v| SYMS.include? k }
26        .to_h
27      case package.delete(:status)
28      when "install ok installed", nil then list << package
29      else list
30      end
31    end
32  end
33
34  def parse_database_file(database)
35    parse_database_content(File.read(database))
36  end
37
38  def parse_database(root='')
39    status_file = File.join(root, 'var', 'lib', 'dpkg', 'status')
40    parse_database_file(status_file)
41  end
```

Listing 6.1 dpkg – Package Database Parsing Code. These methods are used to parse the package database of dpkg.

consider the fields consisting of multiple lines, e.g., `Conffiles` and `Description`. In the next step, the lines are replaced with key-value lists by splitting them again, whereby the key is replaced by a lower case *Symbol*[7]. The format of the resulting list already matches the required format for converting the list to a dictionary. In the final step (Line 25), the list is cleaned up so that only the four fields listed in `SYMS` (Line 17) remain.

The dictionary describing the package is added to the resulting list of packages (returned by the `reduce` method) only if its status stored in the field of the same name is either `install ok installed` – the status of a correctly installed package – or it has no status field at all, i.e., the call to `delete` returns `nil` (Line 28). Otherwise, the dictionary is discarded (Line 29). Note that the status field is removed from the dictionary in any case by the `delete` method, because it is not required anymore after this comparison. The reasoning behind the second condition – no status field at all – is explained in Section 6.4.1.2.

The `openssh-server` package shown in the excerpt of the package database above is shown below in the intermediate format[8] generated by the repository importer. The final format of the Package DB is described in Section 6.4.1.3.

[7] The Ruby language uses Symbols to represent names that are used as constants in a memory efficient way.

[8] The representation of the intermediate format is generated using the `inspect` method.

```
[(...),
 {:package=>"openssh-server", :architecture=>"amd64",
 ⋯:version=>"1:6.0p1-4+deb7u2"},
 (...)]
```

Excerpt from the package database in the intermediate format generated by the Machine Importer showing the openssh-server package.

RPM Package Manager – rpm The package database of rpm is stored in `/var/lib/rpm` and consists of several database files in the Berkeley database format. Since this is a binary format, two theoretical approaches exist to extract the required data: open the database directly using the Berkeley database library to extract the required information or rely on rpm itself to extract the data and convert it into a suitable format. The former approach fails because no public documentation of database structure used by rpm is available. Thus, the latter approach was implemented in the machine importer for rpm databases.

To extract the information from the database, the `rpm` binary was used. It provides the `--query --all` options (short: `-qa`) to output a list of all installed packages and their version identifier. Again, there are two possible approaches: extract the data in situ, i.e., in the running virtual machine, or access the package database files from outside of the dormant virtual machine and extract the data using the `--dbpath` option of `rpm`. On the one hand, the in situ approach requires the virtual machine to be running when the package database should be imported, i.e., for the initial import and after every update of the virtual machine. On the other hand, the approach is guaranteed to work in any case without being susceptible to problems related to different Linux distributions and rpm versions in the virtual machine and the host extracting the data. Using the latter approach, dormant virtual machines can be imported directly by mounting the root directory. Thus, the latter approach is much faster for large numbers of virtual machines. The Update Checker supports both approaches.

The output of `rpm -qa` is a list of installed packages, each package with its full name according to the RPM File Naming Convention [12] (excluding the `.rpm` extension):

```
name-version-release.architecture
```

This convention allows the package names to be easily split into the name, architecture, and the version identifier of the package that consists of the *version* and *release* fields. Unfortunately, *version* and *release* are not the complete version identifier: the *epoch* field is missing. A custom format string has to be set with the `--queryformat` (short: `--qf`) option to make `rpm` output the right format. There is already a preconfigured template that is containing all information the Update Checker needs: `nevra`, an abbreviation for *name*, *epoch*, *version*, *release*, and *architecture*. As this option expects a format string, a record separator has to be set as well, resulting in the complete format string `'%{nevra}\n'`. The resulting output has the following format:

```
name-[epoch:]version-release.architecture
```

The *epoch* is only outputted if it is not `0`, which is the case only for a minority of packages, as the rpm documentation discourages the use of the *epoch* in version numbers. The following snippet shows an exemplary excerpt of the output of the `rpm -qa --qf '%{nevra}\n'` command, showing both packages with and without an *epoch* visible in the output. The package *openssh-server*, epoch *0*, version *6.4p1*, release *5.fc20* for the *x86_64* architecture is highlighted in bold.

```
...
openssh-clients-6.4p1-5.fc20.x86_64
openssh-server-6.4p1-5.fc20.x86_64
openssl-1:1.0.1e-39.fc20.x86_64
...
```

Exemplary excerpt from the package list generated by `rpm -qa --qf '%{nevra}\n'` showing the `openssh-server` package.

The methods shown in Listing 6.2 are responsible for parsing the package list generated by rpm. The structure of the methods is identical to the methods for parsing a dpkg package database (Listing 6.1), except that these methods are part of another class[9]. The method `parse_database_file` is a wrapper that expects the path of a package list. It reads the file and passes the string with the file's content to the `parse_database_content` method that is responsible for the actual parsing (see below for details). Finally, the `parse_database` method automatically builds a default path[10] relative to the given root path and passes it to the `parse_database_file` method.

The parsing of the package list is done using the `reduce` method that is applied to each line (Line 24). A regular expression defined in Line 20 is applied to each line to split the complete package name in the format described above into three components: the package name, the version identifier, and the architecture (Line 25). Note that the version identifier is not split further into its three subcomponents epoch, version, and release. Then, a dictionary containing the package's name, version identifier, and architecture is created and stored in the resulting list of packages, if splitting the complete name was successful (Line 26). When yum is used in combination with rpm, the package database contains a few special entries for signing keys. These entries do not follow the naming scheme and thus the regular expression does not match them. The error handling (Lines 28 to 31) silently ignores these entries[11].

A regular expression is required, because there are almost no restrictions on the allowed characters in the individual components. For example, consider the `openssh-server` and `openssl` packages shown in the excerpt from the package list above. It is not possible to extract the name of the package by splitting at the first dash, because the former contains a dash in its package name. Furthermore, it is also not possible to split at the first colon to extract the package name and the epoch and then split on the last dash to extract the package name, because the former package has no epoch. The

[9] The class definition is not shown because the additional indent would add more line breaks inside statements.

[10] The package list is not part of the regular package database and thus no default path exists. The path `/var/lib/rpm/packages.list` has been chosen arbitrarily.

[11] The exception in Line 29 should never be thrown for valid input data.

```
LIST_REGEX = /^(.*)\-((?:\d+:)?[^-]*\-[^-]*)\.([^.]*)$/
⋮
def parse_database_content(content)
  content.each_line.reduce([]) do |packages, line|
    unless (match = LIST_REGEX.match(line.strip)).nil?
      packages << { package: match[1], version: match[2], architecture:
    match[3] }
    else
      unless line.start_with? 'gpg-pubkey'
        raise "content contains malformed entry, parsing failed: #{line}"
      end
      packages
    end
  end
end

def parse_database_file(database)
  parse_database_content(File.read(database))
end

def parse_database(root='')
  parse_database_file(File.join(root, 'var', 'lib', 'rpm', 'packages.list'))
end
```

Listing 6.2 rpm – Package List Parsing Code. These methods are used to parse the list of installed packages generated by `rpm -qa --qf '%{nevra}\n'`.

`openssh-server` package shown in the excerpt of the package list above is shown below in the intermediate format generated by the repository importer.

```
[(...),
 {:package=>"openssh-server", :version=>"6.4p1-5.fc20",
 ···:architecture=>"x86_64"},
 (...)]
```

Excerpt from the package database in the intermediate format generated by the Machine importer showing the `openssh-server` package.

Repository Configurations

Advanced Packaging Tool – apt The repositories used by apt are configured in the file `/etc/apt/sources.list`. This file contains multiple repository definitions, one per line. Instead of a single `sources.list` file, the repository definitions may also be spread over multiple files in the `/etc/apt/sources.list.d` directory. The format of each definition is as follows:

```
type␣root␣archive␣component␣[component␣...]
```

The *type* field can have the values *deb* or *deb-src* representing repositories containing binary packages and repositories containing source packages, respectively. Only repositories of the former type are relevant to the Update Checker, because source packages are not managed by dpkg at all in terms of bookkeeping of installed source packages. The meaning of the other fields is explained in Section 6.4.1.2. They are required to build the URL of the actual repository that is used to load the list of available packages. An excerpt from a `sources.list` file from a Debian GNU/Linux virtual machine is shown in the following snippet, whereby relevant definitions are highlighted. The methods shown in Listing 6.3 are responsible for parsing the repository configuration.

```
...
deb http://ftp.de.debian.org/debian/ wheezy main
deb-src http://ftp.de.debian.org/debian/ wheezy main

deb http://security.debian.org/ wheezy/updates main
deb-src http://security.debian.org/ wheezy/updates main
...
```

Exemplary excerpt from `/etc/apt/sources.list`.

```
18  SOURCES_REGEX = /^deb\s+([^\s]+)\s+([^\s]+)\s+(.*)$/
 ⋮
43  def parse_config_content(content)
44    content.scan(SOURCES_REGEX).map do |root, arch, comps|
45      comps.split(/\s+/).map do |comp|
46        { :root => root, :archive => arch, :components => comp }
47      end
48    end.flatten
49  end
50
51  def parse_config_file(config_file)
52    parse_config_content(File.read(config_file))
53  end
54
55  def parse_config(root='')
56    base = File.join(root, 'etc', 'apt')
57    result = []
58    result << parse_config_file(File.join(base, 'sources.list'))
59    Dir.glob(File.join(base, 'sources.list.d', '*.list')) do |sources_file|
60      result << parse_config_file(sources_file)
61    end
62    result.reject { |list| list.empty? }.flatten
63  end
```

Listing 6.3 apt – Repository Configuration Parsing Code. These methods are used to parse the repository configuration of apt.

The method `parse_config_file` is a wrapper that expects the path of a single configuration file. It opens this file and passes the string with the file's content to the `parse_config_content` method that is responsible for the actual parsing. The latter method uses the regular expression defined in Line 18 to identify all binary repositories and to split their entries into the corresponding *root* and *archive* fields as well as the remainder of the line containing a list of *component* fields (Line 44). Then, it splits this remainder into individual *component* fields and creates a dictionary with the *root*, *archive*, and *component* fields for each *component* (Lines 45 to 46). Consequently, the inner block creates a list containing multiple dictionaries if a line in the configuration file contains multiple *components*. The `flatten` method is called at the end of the method to convert the list of lists of dictionaries returned by the outer call to `map` into a flat list of dictionaries (Line 48).

The `parse_config` method picks all configuration files relative to a given root path, passes them to `parse_config_file` and adds the resulting lists of dictionaries to the result list (Lines 57 to 61). In a final step, it removes empty lists that are generated if a configuration file contains no valid repositories from the result list and flattens it, i.e., it adds all dictionaries contained in inner lists to the result list itself and removes the inner lists (Line 62).

There is another piece of information that is important for the Update Checker: the architecture of the system, which is required to look up the right repository (see Section 6.4.1.2). This is a piece of information that is built-in to the installed `dpkg` package [34]. It can be retrieved while the virtual machine is running by executing the command `dpkg --print-architecture`. If the virtual machine is not running, the architecture of the dpkg package itself, which matches the system architecture, can be looked up in the package database. Alternatively, the architecture can be manually indicated when importing the virtual machine.

Multiarch setups that enable the installation of library packages from multiple architectures on the same machine are supported by `dpkg`. The most common use case is installing both 64 and 32-bit software in the same virtual machine and having dependencies correctly resolved automatically. In general, libraries for more than one architecture can be installed together and applications from one architecture or another can be installed as alternatives [34]. One popular example of software that requires Multiarch to be enabled in amd64 variant of Debian GNU/Linux is the Android SDK, which is only available in a 32-bit version.

If a Multiarch setup is used by a virtual machine, additional enabled architectures can be queried using the `dpkg --print-foreign-architectures` command while the virtual machine is running. Additionally, there is a file `/var/lib/dpkg/arch` that contains both the base architecture of the system as well as additional enabled architecture. An exemplary `arch` file from a Debian/GNU Linux virtual machine is shown in the following snippet:

```
amd64
i386
```

Exemplary content of `/var/lib/dpkg/arch` for a Multiarch setup.

Yellowdog Updater, Modified – yum The repositories used by yum are configured in multiple `.repo` files in the `/etc/yum.repos.d` directory. The format of those `.repo` files is the *INI* format, which is commonly used for configuration files. It consists of one or more sections, which in turn contain multiple key-value-pairs called properties. Each `.repo` file can contain multiple repository definitions, each defined in its section and consisting of almost a dozen properties. There are three properties relevant to the Update Checker: *baseurl*, which defines the URL of the actual repository, *metalink*, which defines the URL of a service that returns a list of repository mirrors, and *enabled*, which defines whether the repository is enabled. The following snippet shows an excerpt from one of the `.repo` files of a Fedora virtual machine with the relevant properties highlighted. Note that the *baseurl* property is disabled in favor of the more flexible *metalink* approach.

```
[fedora]
name=Fedora $releasever - $basearch
failovermethod=priority
#baseurl=http://download.fedoraproject.org/pub/fedora/linux/releases/
  ↪ $releasever/Everything/$basearch/os/
metalink=https://mirrors.fedoraproject.org/metalink?repo=
  ↪ fedora-$releasever&arch=$basearch
enabled=1
metadata_expire=7d
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-fedora-$releasever-$basearch
skip_if_unavailable=False

[fedora-debuginfo]
...
```

Exemplary excerpt from `/etc/yum.repos.d/fedora.repo`.

Before the individual `.repo` files have been introduced, all repositories were configured in the central `/etc/yum.conf` file that contains the central configuration of yum. Using the sections provided by the INI format, the repository definitions are clearly separated from the central configuration options. This approach is still supported.

The methods shown in Listing 6.4 are responsible for parsing the repository configuration. The method `parse_config_file` is a wrapper that expects the path of a single configuration file. It opens this file and passes the string with the file's content to the `parse_config_content` method that is responsible for the actual parsing. The latter uses the IniParse library to parse the INI format used in the configuration files. It iterates over every section of the file (Line 46) and determines whether the section describes a repository using a few tests (Lines 47 to 48). These tests verify that the section contains an enabled property and one of the *baseurl* or *metalink* properties. If these properties exist, the section very likely describes a repository. The final step is to check whether the enabled property is set to 1. If these tests are satisfied, the method creates a dictionary with the *baseurl* and *metalink* values and adds it to the resulting list of repositories (Lines 49 to 52).

The `parse_config` method picks all configuration files relative to a given root path, passes them to `parse_config_file` and adds the resulting lists of dictionaries to the result list (Lines 63 to 67). In a final step, it removes empty lists that are generated if a configuration file contains no valid repositories from the result list and flattens it (Line 68).

A few variables can be used to make the URLs flexible enough to cover different architectures and releases of a distribution. Table 6.2 lists the available variables and their meanings [126]. It is important to consider the distinction between the base architecture and CPU architecture represented by the `$basearch` and `$arch`

```ruby
CONFIG_FIELDS = %w(baseurl metalink)
⋮
def parse_config_content(config)
  repos = []
  IniParse.parse(config).each do |section|
    next unless section.has_option?('enabled') && section['enabled'] == 1
    next unless section.has_option?('baseurl') || section.has_option?('metalink')
    repos << CONFIG_FIELDS.reduce({}) do |repo, field|
      repo[field.to_sym] = section[field]
      repo
    end
  end
  repos
end

def parse_config_file(config_file)
  parse_config_content(File.read(config_file))
end

def parse_config(root='')
  base = File.join(root, 'etc')
  result = []
  result << parse_config_file(File.join(base, 'yum.conf'))
  Dir.glob(File.join(base, 'yum.repos.d', '*.repo')) do |config_file|
    result << parse_config_file(config_file)
  end
  result.reject { |list| list.empty? }.flatten
end
```

Listing 6.4 yum – Repository Configuration Parsing Code. These methods are used to parse the repository configuration of yum.

variables, respectively. The base architecture is typically used in the URLs of repository definitions, because it describes the architecture of the complete system.

Table 6.2 List of Variables for the Repository Database URL. These variables can be used in the URL of the repository or the *metalink* service used by yum [126]. The first two variables are the most commonly used ones. (The example values are specific for Fedora.)

Variable	Description
`$releasever`	The release number of the installed distribution. Example values: *20*, *19*, *18*, *17*, …
`$basearch`	The base architecture of the system. Example values: *i386*, *x86_64*, *armhfp*, …
`$arch`	The architecture of the CPU. Example values: *i586*, *i686*, *x86_64*, …
`$YUM0` - **`$YUM9`**	Variables that represent the values of environment variables of the same name.

The CPU architecture was used to differentiate between binaries optimized for specific processor generations, but support for processors older than the Pentium Pro (i686) has been dropped in most contemporary distributions, making this distinction obsolete. In the 32-bit editions of Fedora, the base architecture is always *i386*, because it represents the 32-bit architecture in general[12], while the CPU architecture is typically *i686*. On the contrary, there is no such difference between the variables in the 64-bit edition.

There are different ways to determine the specific values used by yum in a virtual machine. If the virtual machine is running, the command `yum version nogroups` can be used to determine both release number and base architecture. Exemplary output of this command from a Fedora virtual machine is shown in the following snippet. The highlighted part contains the required information: release number *20* and base architecture *x86_64*. It can be parsed with a simple regular expression: `/^Installed:\s+(\d+)\/(\w+)/`.

```
Loaded plugins: langpacks, refresh-packagekit
Installed: 20/x86_64  (...)  1238:6b6f949f5a39cafd01c6a8a3ffcb387076a88077
version
```

Exemplary output from `yum version nogroups`.

For dormant virtual machines, the release number can be found in the file `/etc/os-release` that uses the INI format and contains a variety of information. The release number is stored in the *VERSION_ID* property and can be easily extracted using the `IniParse` library. Unfortunately, the base architecture of the system is not stored in an easily accessible file. One way to determine it nevertheless is to rely on the architecture of the `rpm` package itself. However, the architecture of a package is the CPU architecture (`$arch`) and not the base architecture (`$basearch`) that is required to build the correct URLs for the repository. The method `infer_basearch` shown in Listing 6.5 determines the base architecture using a simple heuristic.

```ruby
def infer_basearch(database)
  rpm_arch = find_unique_package(database, 'rpm')[:architecture]
  case rpm_arch
  when /i.86/ then 'i386'
  when /x86_64/ then 'x86_64'
  end
end

def find_unique_package(database, package)
  result = if database.is_a? Array
    rpms = database.select { |pkg| pkg[:package] == package }
  else
    ...
  end
  raise 'package not found / not unique' if result.length != 1
  result[0]
end
```

Listing 6.5 yum – Inferring the Base Architecture. The first method infers the base architecture of a system from the architecture of the `rpm` package. The second method is a helper method to look up a package in a package database that is unique, i.e., existing only in a single architecture.

[12] The Intel 80386 was the first 32-bit processor in the x86 processor family.

A helper method is used to retrieves the `rpm` package from the package database. For databases in the intermediate format this can easily be done using the `select` method (Line 81). The omitted part contains to code to find the package if the database has already been converted to the final format (see Section 6.4.1.3). The method ensures that only a single package is found and raises an exception otherwise (Line 92). Alternatively, the base architecture and release number can be manually set when the virtual machine is imported.

Another approach that works for running virtual machines is to ask `yum` for the complete URLs of the repositories using `yum repoinfo`. In this case `yum` takes care of replacing the variables before printing the URLs, sparing the Update Checker from determining the values manually. The following snippet shows an exemplary excerpt from the output of this command on a 64-bit virtual machine running Fedora 20. It shows information regarding the base repository of Fedora, with the important parts highlighted, and the first lines of the information regarding the repository used for updates.

```
Loaded plugins: langpacks, refresh-packagekit
Repo-id      : fedora/20/x86_64
Repo-name    : Fedora 20 - x86_64
(...)
Repo-size    : 38 G
Repo-metalink: https://mirrors.fedoraproject.org/metalink?
···repo=fedora-20&arch=x86_64
  Updated    : Fri Dec 13 09:55:41 2013
Repo-baseurl : http://mirror2.atrpms.net/fedora/linux/releases/20/
···Everything/x86_64/os/ (92 more)
Repo-expire  : 604.800 second(s) (last: Mon Oct 13 10:27:43 2014)
Repo-filename: /etc/yum.repos.d/fedora.repo

Repo-id      : updates/20/x86_64
Repo-name    : Fedora 20 - x86_64 - Updates
...
```

Exemplary excerpt from the output of `yum repoinfo`.

For each repository, the output contains a few lines with different information in a key-value format. A blank line separates multiple repositories. The methods shown in Listing 6.6 parse the output and extract the *metalink* and *baseurl* properties. The first method does the actual parsing. First, it splits the output into separate blocks for every repository and uses the `reduce` method on the resulting list of blocks (Line 97). For each block, it extracts (a part of) the name and the value of the two desired properties, whereby it has to omit the text in parentheses after the URL in the *baseurl* property (Line 98). In the next step, it creates a dictionary with these properties using symbols corresponding to the partial names of the properties as keys and adds the dictionary to the resulting list of repositories unless it is empty, i.e., the block contained no repository[13] (Line 100). The second method again is only a wrapper that reads a file at a given path and passes the file's contents to `parse_repoinfo_content`.

[13] The final block (not shown in the excerpt) contains the number of packages available in all repositories, but no repository.

```
def parse_repoinfo_content(repoinfo)
  repoinfo.split(/\n\n/).reduce([]) do |list, repo|
    data = repo.scan /Repo-(metalink|baseurl)\s*:\s+([^ \n]+)/
    unless data.empty?
      list << data.map { |key, value| [key.to_sym, value] }.to_h
    else
      list
    end
  end
end

def parse_repoinfo(repoinfo_file)
  parse_repoinfo_content(File.read(repoinfo_file))
end
```

Listing 6.6 yum – Repository Configuration Parsing Code. These methods are used to parse the repository configuration of yum.

Multiarch setups are implemented in a much simpler way in yum/rpm compared to apt/dpkg. For the most common scenarios like 32-bit software on x86_64 machines the libraries for the secondary architecture are included in the repository for the base architecture. For example, the standard x86_64 of Fedora 20 contains 15.054 x86_64 packages, 17.341 noarch packages, and 6.112 i686 packages. Libraries for the secondary architecture can be installed without any changes to the repository configuration by simply appending the architecture to the package name like in `yum install glibc.i686`, which installs the 32-bit version of the GNU C library.

6.4.1.2 Repository Importer

The repository importer is responsible for importing the list of packages available in a repository into the Repository Cache. Like the machine importer, the repository importer has to parse the information retrieved from the repository and remove extraneous information before the data is stored in the Repository Cache. The format used by the Repository Cache is described in Section 6.4.1.3.

Each of the considered package management systems uses a database in a distinct format to keep track of the versions of packages available at a repository. A repository importer tailored to the specific format of the package management system is used to gather the required information and store it in the Repository Cache. In the following, the format of the databases is described and corresponding code to parse the information is presented.

Advanced Packaging Tool – apt The repository database of an apt repository can be found using the following URL. The fields used to build the URL are described in Table 6.3. The values for all fields are taken from the repository configuration.

```
root/dists/archive/component/binary-architecture/Packages.type
```

Table 6.3 Parts of the repository database URL. These fields are used to build the URL of the package database used by apt. (The example values are specific for Debian GNU/Linux. The *archive* and *component* names used in Debian derivatives may vary.)

Field	Description
root	The root URL of the repository or mirror.
archive	Used to divide a repository for different releases. Example values: *stable*, *stable-updates*, *testing*, …
component	Used to subdivide a repository by license type and level of support. Example values: *main*, *contrib*, *nonFree*, …
architecture	Used to further subdivide a repository in sections for different architectures. Example values: *i386*, *amd64*, *armhf*, *mips*, *sparc*, *s390*, *all*, …
type	The compression format of the package list. Example values: *gz*, *bz2*

An exemplary URL of an apt repository is shown below. The values for the fields are taken from the configuration shown above, except for the format that has been randomly chosen.

```
http://ftp.de.debian.org/debian/dists/wheezy/main/binary-amd64/Packages.bz2
```

For Multiarch setups, the repositories for all enabled architectures have to be imported separately. The Scan Engine (see Section 6.4.1.4) uses the architecture stored with each package to determine the right package list to use for version comparisons. It should be noted that there is no distinct repository for architecture-independent packages, but those packages are available in every repository.

The repository database available at the aforementioned URL uses the same format as the package database of dpkg, except that packages in the repository database do not have a status field. The database can nevertheless be parsed using the method that is used for parsing the package database (see Section 6.4.1.1), because that method has been implemented with this peculiarity in mind. The second condition in Line 28 ensures that all packages in the repository database are added to the list of packages.

Yellowdog Updater, Modified – yum Looking up the repository database of a yum repository is a two-step process:

1. Determine the URL of a specific repository using a service available at the URL pointed to by the *metalink* property of the repository definition. This step was introduced in recent versions of yum to determine the nearest repository based on the requested release number, base architecture, and the geolocation of the client. Earlier versions of yum used the *baseurl* property of the repository definition to store the specific URL of the repository and thus required no additional lookup step. If the *baseurl* is already known, e.g., from parsing the output of `yum repoinfo`, this step can be omitted.

2. Determine the URL of the repository database for a specific repository by extracting its path from a file containing the metadata of all databases available at the repository. In contrast to apt, yum provides several database files on each repository. The package database itself is available in multiple formats and there are databases to store additional data like changelog information that is not relevant to the Update Checker.

The *metalink* URL used in Step 1 points to an XML document containing a list of mirrors together with a preference rating to select the most suitable mirror depending on the geolocation of the client. The next snippet shows an excerpt from that XML document returned for release number *20* and base architecture *x86_64* for the base repository of Fedora. The preferred HTTP mirror is highlighted.

```xml
<?xml version="1.0" encoding="utf-8"?>
<metalink version="3.0" xmlns="http://www.metalinker.org/" (...)>
  <files>
    <file name="repomd.xml">
      <mm0:timestamp>1386924941</mm0:timestamp>
      <size>3847</size>
      <verification>
        (...)
        <hash type="sha512">d6850133(...)66085b2e</hash>
      </verification>
      <resources maxconnections="1">
      <resources maxconnections="1">
        <url protocol="ftp" (...)>ftp://(...)</url>
        <url protocol="rsync" (...)>rsync://(...)</url>
        <url protocol="http" type="http" location="CZ" preference="100">
        ···http://mirror.karneval.cz/pub/linux/fedora/linux/releases/20/
        ···Everything/x86_64/os/repodata/repomd.xml</url>
        <url protocol="http" (...)>http://(...)</url>
        (...)
      </resources>
    </file>
  </files>
</metalink>
```

Exemplary excerpt from the XML document returned when accessing the *metalink* URL.

The method `parse_metalink` shown in Listing 6.7 parses the contents of the XML document retrieved using the *metalink* URL. First, it uses an *XML Path Language* (XPath) [171] expression to extract all *url* nodes from the document. Additionally, the protocol is restricted to HTTP (Line 115)[14]. Although it would be possible to extract the URLs directly by appending `/text()` to the expression, this approach is preferred because it allows to extract the attribute values as well. Then, it sorts the mirrors by preference an reverses the list, because the highest priority repository should be the first element (Line 116)[15]. Finally, it uses the `map` method to replace the nodes with their corresponding URLs for easier access by the caller (Line 117) and returns the result to the caller. A list of mirrors is returned instead of a single mirror, so

[14] Additional restrictions can be added, e.g., to only select mirrors with a preference above 90.
[15] The list of repositories is very likely already sorted, thus the manual sorting is just a precaution.

that the Update Checker can switch to another repository if the one with the highest preference fails.

```ruby
def parse_metalink(metalink_url)
  open(metalink_url) do |metalink|
    Nokogiri::XML(metalink)
      .xpath('/xmlns:metalink/xmlns:files/xmlns:file/xmlns:resources/'\
     'xmlns:url[@protocol="http"]')
      .sort_by { |mirror| mirror['preference'].to_i }.reverse
      .map { |mirror| mirror.text }
  end
end
```

Listing 6.7 yum – Parsing the *metalink* Document. This codes parses the XML document containing a list of mirrors for a given repository, release and architecture.

The URLs in the mirror list directly point to the document containing the metadata of the different database. Because the name of this document is known (see the snippet below), the *baseurl* can be extracted from that URL by removing the trailing `repodata/repomd.xml`. In the same way, the URL of the document can be constructed if the *metalink* property is not set and the older approach is used. In this case, `repodata/repomd.xml` has to be appended to the *baseurl*.

```
baseurl/repodata/repomd.xml
```

In Step 2, the XML document with the repository metadata named `repomd.xml` is retrieved using the URL that has been either extracted from the mirror list or constructed using the *baseurl* property. It contains several `<data>` tags that contain the URL, checksums, and metadata of the diverse databases available at the repository. The repository database required by the Update Checker is marked with the `type="primary"` attribute. This entry contains a child tag `<location>` that in turn contains the path of the actual database relative to the *baseurl*. The following snippet shows the contents from that XML document returned for release number *20* and base architecture *x86_64* by the base repository of Fedora. The important part, i.e., the location of the repository database, has been highlighted.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<repomd xmlns="http://linux.duke.edu/metadata/repo" (...)>
 <revision>1386924430</revision>
 <tags>(...)</tags>
  <data type="group">(...)</data>
  <data type="filelists">(...)</data>
  <data type="group_gz">(...)</data>
  <data type="primary">
    <checksum type="sha256">d7777ea6(...)c4e82094</checksum>
    <open-checksum type="sha256">18f77a31(...)292edb3c</open-checksum>
    <location href="repodata/d7777ea6(...)c4e82094-primary.xml.gz"/>
    <timestamp>1386924704</timestamp>
    <size>10148965</size>
    <open-size>87121294</open-size>
  </data>
  <data type="primary_db">(...)</data>
  <data type="other_db">(...)</data>
  <data type="other">(...)</data>
  <data type="filelists_db">(...)</data>
</repomd>
```

Exemplary content from the `repomd.xml` document.

The method that parses this document is shown in Listing 6.8. It extracts the relative path of the repository database using an XPath expression.

```ruby
def parse_repo_metadata(repomd_url)
  open(repomd_url) do |repomd|
    Nokogiri::XML(repomd)
      .xpath('string(/xmlns:repomd/xmlns:data[@type="primary"]/'\
     'xmlns:location/@href)')
  end
end
```

Listing 6.8 yum – Parsing the Metadata Document. This method extracts the relative path of the repository database from the XML document containing metadata about the different databases of a repository.

The URL of the package database is then build be built by concatenating the extracted relative path determined and the *baseurl*. Using the values from the last snippets, this leads to the following URL:

```
http://mirror.karneval.cz/pub/linux/fedora/linux/releases/20/Everything/
···x86_64/os/repodata/d7777ea6(...)c4e82094-primary.xml.gz
```

After the URL of the repository database has been determined, it can be retrieved and parsed. The database is an XML file that contains a `<package>` tag for each available package, which in turn contains the `<name>`, `<arch>` and `<version>` tags providing the information required by the Update Checker. The following snippet shows an excerpt from the repository database of the Fedora base repository for release number *20* and base architecture *x86_64*[16]. The parts used by the Update Checker are highlighted.

```
...
</package>
<package type="rpm">
  <name>openssh-server</name>
  <arch>x86_64</arch>
  <version epoch="0" ver="6.3p1" rel="5.fc20"/>
  <checksum type="sha256" pkgid="YES">eec94c2a(...)1e6ab769</checksum>
  <summary>An open source SSH server daemon</summary>
  <description>OpenSSH is a free version of SSH(...)</description>
  <packager>Fedora Project</packager>
  <url>http://www.openssh.com/portable.html</url>
```

[16] Note that the version of the `openssh-server` package available at the repository is older than the installed version shown in previous snippets, because the base repository contains the initial version of the package at the time of the release and the virtual machine has been updated before importing it into the Update Checker.

```
  <time file="1383328660" build="1383323983"/>
  <size package="375688" installed="756218" archive="758628"/>
  <location href="Packages/o/openssh-server-6.3p1-5.fc20.x86_64.rpm"/>
  <format>(...)</format>
</package>
<package type="rpm">
  <name>openssh-server-sysvinit</name>
...
```

Exemplary excerpt from the repository database.

The method shown in Listing 6.9 parses the repository database file.

```ruby
def parse_repo_database(repodb_url, compression=:none)
  open(repodb_url) do |repodb|
    repodb = case compression
      when :gzip then Zlib::GzipReader.new(repodb)
      when :none then repodb
    end
    xml = Nokogiri::XML::Reader(repodb.read)

    database = []
    package = nil
    capture = false
    text = nil

    xml.each do |node|
      case node.node_type
      when Nokogiri::XML::Reader::TYPE_ELEMENT
        case node.name
        when 'package' then package = {}
        when 'name', 'arch' then capture = true
        when 'version'
          attr = node.attributes
          package[:version] =
            (attr['epoch'] != '0' ? attr['epoch'] + ':' : '') +
            attr['ver'] + '-' + attr['rel']
        end
      when Nokogiri::XML::Reader::TYPE_TEXT
        text = node.value if capture
      when Nokogiri::XML::Reader::TYPE_END_ELEMENT
        case node.name
          when 'name' then package[:package] = text
          when 'arch' then package[:architecture] = text
          when 'package' then database << package
        end
        capture = false
      end
    end
    database
  end
end
```

Listing 6.9 yum – Repository Database Parser. This method extracts the version information of all available packages from the repository database and returns a list of packages in the intermediate format that is also used by Machine Importers.

All approaches to parse this database using an approach based on the *Document Object Model* (DOM) [172], thus also the XPath approach used to extract URLs in the first two steps, are inappropriate. The database that is shown in excerpts above has a size of about 85 MiB at 38,500 entries and requires almost 950 MiB of memory when opened using Nokogiri's DOM parser. Typically, a parser based on the *Simple API for XML* (SAX) de facto standard [135] is used for larger documents like this. The advantage over a DOM parser is that it parses the XML document sequentially and thus it never has to reside in memory completely. This simultaneously is the biggest drawback of this kind of parsers: the document is delivered to the parsing code in smallest possible pieces, e.g., opening and closing XML tags or text nodes. If any kind of state is required for parsing the document, it has to be kept manually, because SAX does not allow rewinding the document or generating a callback a second time.

The Nokogiri library additionally provides a middle course using the `Reader` class. It does not generate different callbacks for different types of nodes, but instead loops over all elements of the XML document. This class is used to parse the repository database in the `parse_repo_database` method shown in Listing 6.9. It is able to open both uncompressed and *gzip* compressed repository databases (Lines 130 to 133), although support for other compression formats can be added easily.

The main parsing is done in the loop in Lines 141 to 163. An outer `case` construct is used to distinguish nodes by their type, whereas inner `case` constructs are used in case of XML tags to further distinguish them by their name. An opening *package* tag, i.e., a node of the `TYPE_ELEMENT` type with the corresponding name, marks the start of the next package. A new dictionary is thus created in this case (Line 145). The *name* and *arch* tags set a flag that captures the content of text nodes (Line 146). The *version* tag contains three attributes with the components of the version identifier. The method generates a version identifier in the format described in Section 6.3.1 that is used throughout the Update Checker and stores it in the dictionary (Lines 148 to 151). The content of text nodes is stored in a temporary variable if the `capture` flag is set.

A closing *name* or *arch* tag ends the package or architecture name, respectively. The text captured in the temporary variable is stored in the dictionary. Finally, the closing *package* tag ends the description of the package. At this point, the dictionary is added to the database. The result of this method is a list of dictionaries describing packages: the intermediate format that is also generated by Machine Importers (see Section 6.4.1.1).

Surprisingly, the SAX approach is about 125 % slower than the approach presented above in benchmarks that have been done during the implementation of the parser, whereas the DOM approach is 25 % faster. However, the improved performance of the latter does not compensate for the huge memory requirements.

6.4.1.3 Internal Databases and Caches

The Update Checker uses four different databases and caches to store information about machines and repositories as well as its findings (see Figure 6.2). The techniques and formats used to store this information are described in detail in this section.

The biggest amount of information is the list of installed packages for each of the registered virtual machines in the Package DB and the list of available packages at each of the repositories in the Repository Cache. In both cases, for each package, the name, version and architecture have to be stored. The need to store the architecture for each package individually is a result of the support for Multiarch installations: the Scan Engine needs to know which repository to check for updated versions of any given package. Therefore, each installed or available package is represented by a name-version-architecture triple.

Initially, it was planned to store this information in a database together with the other data the Update Checker requires. Unfortunately, importing a virtual machine or updating the list of available packages of a repository is very slow using a database. Whether this is caused by the database itself (*SQLite3*) or the Object-Relational Mapping (ORM) library used (*DataMapper*) has not been examined in detail.

Instead, the Update Checker stores both the Package DB and the Repository Cache by serializing the internal data structures in the JavaScript Object Notation (JSON) [30]. An individual file is used per virtual machine or repository. In a test conducted during the implementation of the Update Checker, the JSON approach was faster by a factor of more than 23 compared to the database approach[17].

Both Machine Importers and Repository Importers return the imported databases in an intermediate format: as a list of dictionaries describing individual packages. Each dictionary includes a package's name, version identifier, and architecture. While this format is sufficient for the Package DB, because the Scan Engine just iterates over all installed packages of a virtual machine sequentially to check for outdated software, it is inadequate with respect to the Repository Cache, because the Scan Engine has to look up packages by their name and architecture. A two-level dictionary as shown below is the optimal data structure for the Repository Cache. It maps from a package's architecture to an inner dictionary that in turn maps from the package's name to its version. It is important to note that a single-level dictionary mapping from a package's name to (a dictionary or list storing its) version and architecture does not work with Multiarch setups, because package names are not unique in yum package databases.

```
{"amd64"=>{(...), "openssh-server"=>"1:6.0p1-4+deb7u2",                ⤸
          ···"openssl"=>"1.0.1e-2+deb7u12", (...)},
 "all"=>{(...), "openssh-blacklist"=>"0.4.1+nmu1", (...)}}
```

Optimal data structure for the Repository Cache.

This data structure is also used for the Package DB, because it saves storage space for the package databases when stored as JSON files. For an exemplary package database of 1.140 packages, this format saves 60 % of space, because the keys, i.e., `:package`, `:version`, and `:architecture`, are stored implicitly and the architecture, e.g., `amd64` or `all`, are not stored repeatedly for every package. It is created from the intermediate format using the function shown in Listing 6.10.

[17] The time required for importing two Debian repository databases was compared. The import took 2.16 seconds using the JSON serialization compared to 50.02 seconds using the database.

```
13 def two_level_database(database)
14   database
15     .group_by { |pkg| pkg[:architecture] }
16     .map do |arch, list|
17       [arch, list.map { |pkg| [pkg[:package], pkg[:version]] }.to_h]
18     end
19     .to_h
20 end
```

Listing 6.10 Data Structure Conversion. This function converts from the intermediate format generated by both Machine Importers and Repository Importers to the data structure used by the Package DB and Repository Cache.

It uses the group_by method to create a dictionary that maps from the architecture to a list of dictionaries representing all packages having this architecture (Line 15). The format of the dictionaries is not modified in this step compared to the intermediate format. The inner block in Line 17 converts this list of dictionaries into a dictionary mapping from a package's name to its version using a combination of the map and to_h methods. An outer call to the map method is used to apply the conversion in the block to the lists of all architectures (Line 16). In the last step, the list of lists returned by map is converted into a dictionary again (Line 19). This final conversion is the reason why the inner block does not only return the desired dictionary, but a list of the architecture and the corresponding dictionary.

Information about outdated packages and available updates is stored in the Result Cache to improve the response time for subsequent requests. For each outdated package the following information is stored in a dictionary: the name of the package, its architecture, the installed version, the latest version, the repository providing the latest version, and the priority of the update. A list of outdated packages is stored for every registered virtual machine after it has been scanned for the first time. Like the Package DB and Repository cached, the information stored in the Result Cache is serialized using JSON and stored in an individual file per virtual machine. An exemplary excerpt of such a list is shown below [18].

```
[(...),
 {:package=>"openssl", :architecture=>"amd64",                          ⤸
 ···:installed=>"1.0.1e-2+deb7u12", :latest=>"1.0.1e-2+deb7u17",         ⤸
 ···:repository=>"54b16a91-ec30-4c8b-abe0-cbd870c768ea", :priority=>1},
 (...)]
```

Excerpt from a list of results in the Result Cache showing the openssl package.

The Metadata DB is the most important database of the Update Checker, because it is the link between the individual JSON files described above. Internally, each registered virtual machine and repository is assigned an internal identifier. This identifier is used as the file name for the JSON files that build the Package DB, Repository Cache, and Result chance, and thus it associates the files with the represented entity. Additionally, the Metadata DB also stores a list of enabled repositories for each virtual machine

[18] No update was available for the openssh-server package used as an example throughout this section. The openssl package (already included in the last snippet) is shown instead.

and metadata further describing it, e.g., its architecture and release (the repository configuration), information about the owner to facilitate automatic notifications via email about important findings, and timestamps of virtual machine imports, repository cache updates, or scans for available updates that are important for expiring cached information and thereby the for ensuring the currentness of the stored data. Contrary to the other databases and caches above, the Metadata DB is implemented using a SQLite3 database instead of JSON files.

6.4.1.4 Scan Engine

The scan engine is the central component that scans a virtual machine for outdated packages. This section describes the implementation of the actual scan process, the interpretation of scan results, and the version comparison logic.

Scan Process

The `scan` method shown in Listing 6.11 is implementing the scan process. It is only invoked when results are not available in the Result Cache[19]. This method consists of three nested “loops” that are implemented using the `map` and `reduce` methods.

```ruby
18 def scan(vm)
19   vm.packages.map do |arch, package_version|
20     package_version.map do |package, version|
21       repos = vm.repositories.reduce([]) do |result, repository|
22         if repository.packages.include? arch
23           repository_version = repository.packages[arch][package]
24           unless repository_version.nil? || repository_version == version
25             if Version.new(repository_version) > Version.new(version)
26               result << repository
27             end
28           end
29         end
30         result
31       end
32       repos.empty? ? nil : finding(package, arch, version, repos)
33     end
34   end.flatten.compact
35 end
```

Listing 6.11 Scan Engine: Scan Process. This method implements the scanning process that determines outdated packages in virtual machines and the available updated versions available in one or more of the repositories.

The innermost “loop” (Lines 21 to 31) returns a list of repositories that contain newer version for a single package passed in the block parameters `arch`, `package`, and `version` of the surrounding “loops” in Lines 19 and 20, respectively. It iterates over all repositories of the virtual machine and checks whether the repository contains packages of the desired architecture. This is necessary because not every repository

[19] The corresponding logic that also has to ensure the Repository Cache is up-to-date before it calls `scan` is not shown.

contains packages for every architecture – `dpkg` uses separate repositories for each architecture. If the repository contains any package of the desired architecture, the method tries to retrieve the version of the desired package.

Only if the package is found and its version identifier is not equal to the version identifier of the installed package – tested using a simple string comparison (Line 24), the method uses a library for version comparison (see below) to determine if the repository contains a newer version of the package (Line 25) and adds the repository to the resulting list. Note that the package available in the repository might also be older, so the string equality test is not sufficient.

The middle "loop" (Lines 20 to 33) iterates over all key-value pairs of a dictionary mapping from the packages of a single architecture to their corresponding versions[20] Both the architecture and the dictionary are passed in the block parameters of the surrounding "loop" in Line 19. For each element, the innermost "loop" is executed. If it returns one or more repositories, the result of calling the `finding` method (shown below) is added to the resulting list of findings. The package's name, architecture, installed version and a list of repositories with newer package versions are passed as arguments to that method.

Finally, the outer "loop" (Lines 19 to 34) iterates over the set of architectures of all packages installed in the virtual machine and executes the middle "loop" for this architecture. The resulting list of lists (a list of lists of findings) is flattened and all `nil` values are removed (Line 34) before it is returned to the caller.

The `finding` method shown in Listing 6.12 creates a dictionary in the format described in the last section (Lines 50 to 52). This dictionary contains information about a single outdated package and is added to the result list of the scan process. It uses the package's name, architecture and installed version that are passed as arguments for the first three fields of the dictionary. For the remaining three fields, it has to determine the repository that contains the latest version of a package and has the highest priority (smaller priority value). This is trivial for a single repository containing a newer version of the package (Line 40), whereas it is a multi-step process for more than one repository.

First, the latest available version of the package is determined using the `sort_by` method with the library for version comparison. After sorting, the last repository in the list contains the latest version of the package (Lines 42 to 44). What is left is to determine the priority of the update. In the current version, this is done by assigning each repository a priority using a rudimentary heuristic: packages provided by repositories containing the word "security" in their URL are considered as high priority updates, whereas packages from other repositories are considered as low priority updates. The priority is expressed using the integer values `1` and `2`, respectively. Unfortunately, the priority is not considered by the `sort_by` method.

In the next step, all repositories providing the latest version of the package are determined using a simple string comparison (Line 46). The list of repositories is then

[20] The format of the package database is shown in the second last snippet showing the "Optimal data structure for the Repository Cache".

```
37 def finding(package, arch, version, repos)
38   repo =
39     case repos.length
40     when 1 then repos.first
41     else
42       latest_version = repos
43         .sort_by { |repo| Version.new(repo.packages[arch][package]) }
44         .last.packages[arch][package]
45       repos
46         .select { |repo| repo.packages[arch][package] == latest_version }
47         .sort { |repo_a, repo_b| repo_a.priority <=> repo_b.priority }
48         .first
49     end
50   { package: package, architecture: arch, installed: version,
51     latest: repo.packages[arch][package], repository: repo.id,
52     priority: repo.priority }
53 end
```

Listing 6.12 Scan Engine: Creation of Result List Entries. This method is responsible for creating dictionaries in the correct format for the result list of the scan process.

sorted by their priority (Line 45) and the repository with the highest priority – the lowest value and thus the first element – is selected (Lines 47 and 48). The latest version of the package and the selected repository's internal identifier and priority are then added to the dictionary.

Interpretation of Results

The raw list of outdated packages contains detailed information about the outdated packages. However, for a quick examination this list contains too much information. It can be easily condensed into meaningful numbers or simple lists. For example, the number of outdated packages can be determined using `vm.findings.length`, whereas a list of outdated packages can be retrieved using `vm.findings.map { |finding| finding[:package] }`.

More meaningful than the total number of outdated packages or their names might be the number of outdated packages grouped by priority. This information can be retrieved using the `update_count` method shown in Listing 6.13. First, the list of outdated packages is grouped by the priority assigned to the update. Then, the number of updates is determined for each priority and a hash.

Version Comparison

One particular problem discovered during the implementation of the Update Checker is the format of the version numbers used by the different package management systems and distributions, respectively. While most of the distributions use versions composed of the fields epoch, version and release, there are subtle differences between the distributions, e.g., separators, format of the release field, etc. Even the *versionomy*

```
def update_count(vm)
  vm.findings
    .group_by { |finding| finding[:priority] }
    .map { |priority, list| [priority, list.length] }
    .to_h
end
```

Listing 6.13 Scan Engine: Number of Updates grouped by Priority. This method groups the available updates by priority and calculates the number of updates for each priority.

gem, a Ruby library especially designed for version comparisons, failed to correctly compare Debian version numbers.

The most obvious method is to use the dpkg binary itself to perform the version comparison. Its command line interface provides an option for this specific use case: `--compare-versions`. The drawback of this approach is the creation of a new process for each comparison. This method can therefore not be used in the Update Checker when the potentially high number of comparisons for checking even a single virtual machine is considered.

The Update Checker uses a Ruby library named *dpkg-ruby*[21] that provides a native extension for comparing versions. There even exists an older version of the library that implements the comparison algorithm directly in the Ruby language without relying on the native extension. Both versions of the library can be used with the Update Checker. The Ruby-only version has the advantage of keeping the set of dependencies smaller and is thus preferred, although it might be slightly slower compared to the version using the native extension[22]. To compensate the potential loss of performance, a simple string comparison is performed before the actual version comparison is done (see Listing 6.11, Line 24). Only if the string comparison finds differences, the actual version comparison algorithm is used. The preceding string comparison thus reduces the number of times the full-blown version comparison algorithm is invoked. Additionally, the string comparison is much more efficient than the version comparison algorithm, which splits the version into its individual components (epoch, version, and release) and compares those individually, whereby it considers the peculiarities of some versioning schemes. The savings for packages that are not outdated should outweigh the additional comparison for outdated packages, because in most cases only a fraction of the packages are outdated. The library was slightly modified to make it compatible to all version numbers encountered in Debian and Fedora during the implementation of the Update Checker.

6.4.1.5 Remote Importer

The Remote Importer that has been developed for easy registration of virtual machines and updating of the information about these virtual machines. It gathers the required

[21] Newer versions are called *ruby-debian*. This library seems to be developed by people from Debian, but there is no real homepage for this library, except for an entry in Debian's package database.
[22] No performance evaluation was done to confirm this guess.

information for the import or update process directly in the respective virtual machine using a Data Collector, compresses the information and transfers it to the Update Checker. Finally, it initiates the registration of the virtual machine or the update of the stored data regarding the virtual machine, respectively. The information gathered by the Remote Importer includes package databases and repository configuration files. Depending on the package management system, this information might also be the output of package management tools that is captured in files. The Data Collectors provide an abstraction of the package management system, so the Remote Importer can be used on with different existing package management systems and easily adapted to new ones. Implementations for the apt/dpkg and yum/rpm package management systems are provided.

To interact with the update checker, the Remote Importer uses the XML-RPC API provided by the Update Checker. This API provides the same set of features as the command line interface that is provided for local usage. Both the command line interface and the API can be used to query the number of available updates for a single or multiple virtual machines or the list of outdated packages of a single virtual machine. Furthermore, registering new virtual machines or updating the stored information of virtual machines already registered is possible using both interfaces. The XML-RPC API also facilitates the easy integration of the Update Checker with existing virtual machine management systems, e.g., the ICS and the XGE.

6.4.2 Centralized Update Process

The tools for merging of the package database and detection of relation incompatibilities between the updated base layer and a user layer are implemented in the Python programming language. The implementation is tailored to the Debian Package Manager `dpkg`, which is also used by other distributions based on Debian[23]. The description starts with the determination of the relevant parts of the package database. Afterwards, the merge algorithm for package databases is described. The section ends with a description of the algorithm that detects relation incompatibilities.

6.4.2.1 Relevant Parts of the Package Database

The package database of `dpkg` is stored in `/var/lib/dpkg` and consists of several parts. For the task at hand, the most relevant part is the `status` file that is the central part of the package database and contains the metadata for each package installed on the system. Its format is described in the context of the Update Checker's Machine Importer (Section 6.4.1.1). The merge process of the `status` file is described in the next section.

Additional parts of the package database that are relevant for the centralized update process are the `info` directory contains file lists, checksums, lists of configuration files, and installation/removal scripts for every package as well as the `diversions`

[23] Support for other package management systems is possible by implementing a small compatibility layer that enables reading and writing of the respective package databases.

file that contains records about files from one package that have been replaced by files from another package. This enables preserving the changed files when the original package is updated.

The `info` directory contains a set of files for each package that stores the information listed above. This part of the package database must not be merged, because it always represents the actual packages visible to the system. Because of the union mount semantics, an older package installed in the user layer always hides the corresponding package in the base layer, even if it is newer. In this case, the information in the `info` directory is nevertheless correct, although this is an instance of a masking incompatibility that will be detected by the merge process.

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 a rare event during an update, it might occur when new packages need to be installed within the scope of an update. This file has to be merged using a simple algorithm that is driven by the semantics of the union mounts. The merged version of the file is a copy of the version in the user layer with all entries from the base layer's version added that have no counterpart in the user layer's version, i.e., in case of conflicting entries the entry from the user layer is chosen. This corresponds to the situation in the file system: the replacement file in the user layer hides the replacement file in the base layer.

6.4.2.2 Merging the Package Database

In the following, the term package database is used to refer to the `status` file and the term entry refers to the entire metadata of a single package that consists of a set of fields as described in Section 6.4.1.1.

The inputs of the merge algorithm are package databases from the old base layer, the (updated) base layer, and the user layer. It works by determining all packages that are updated, added or removed in the base layer. These packages are identified by comparing corresponding entries in the package databases of the base layer and the old base layer. Then, it copies all entries that fall into none of the three categories from the user layer's package database to the resulting package database.

For each of the entries in the three categories, the algorithm does a series of checks to determine which entry has to be copied to the resulting package database: the entry from the user layer's or the base layer's package database. Additionally, these checks described below discover masking incompatibilities.

Figure 6.11 shows the flow chart for packages updated in the base layer. The first step is to determine whether the entry in the user layer's package database has been modified compared to the one in the old base layer. If the entries are equal, the files of that package exist only in the base layer and thus are visible in the composite disk image. In this case, the base layer is compatible and the entry from the base layer's package database is copied to the resulting package database.

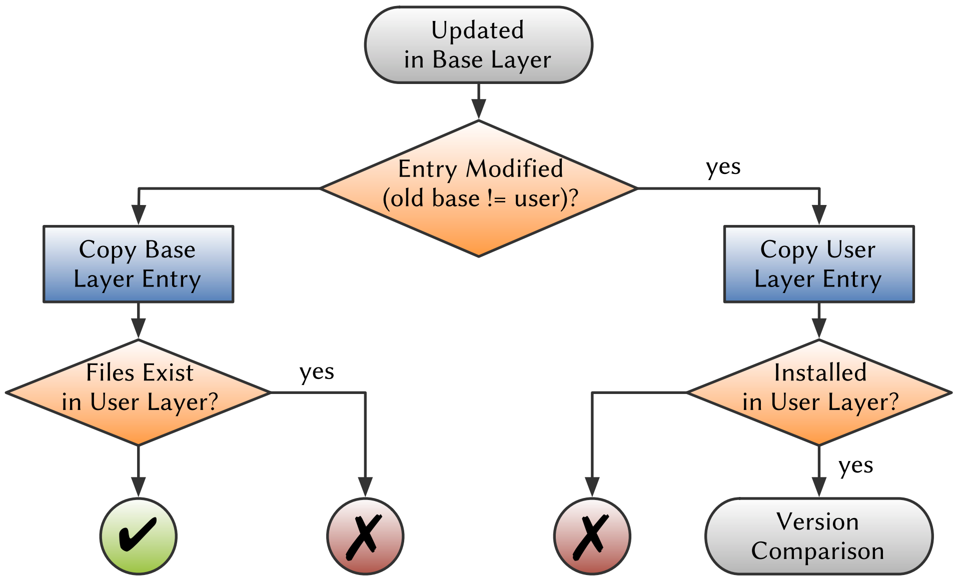


Figure 6.11 Required Checks for Updated Packages. This flow chart shows a number of tests that have to be performed for updated packages.

However, there are two rare cases in which the updated files are not visible: If the user uninstalled the package, the files are hidden by corresponding whiteout files in the user layer. Furthermore, the user might have reinstalled the package afterwards using the same version as the one installed in the old base layer. Both cases are instances of a masking incompatibility. In order to detect this incompatibility, the existence of either real files belonging to the package or corresponding whiteout files in the user layer has to be verified. The most simple approach is to check if a file list of that package exists in the `/var/lib/dpkg/info` directory of the user layer. If this file exists, the package is installed and the base layer is incompatible.

In case of a modified entry, the package has either been updated, downgraded or removed. In all three cases, the updated files in the base layer are hidden in the composite disk image, either by real files belonging to the package or corresponding whiteout files. The entry from the user layer's package database is copied to the resulting package database. If the package was removed, the base layer is definitely incompatible, whereas the installed versions need to be compared otherwise to determine the compatibility of the base layer. This comparison is described in a separate flow chart below.

Figure 6.12 shows the flow chart for version comparison. The versions of the packages installed in the base layer and the user layer are compared using the dpkg binary with the `--compare-versions` option. If the version installed in the base layer is greater than the version installed in the user layer, the base layer is incompatible, because the modified files are hidden by their respective older versions in the user layer. Otherwise, the base layer is compatible irrespective of the hidden files, because the version in the user layer is has at least the same version as the one in the base layer.

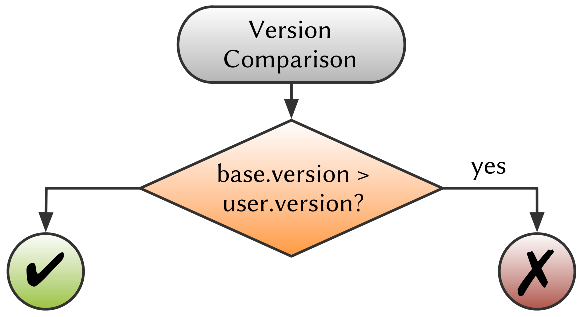


Figure 6.12 Version Comparison Check. This flow chart shows the test required to determine if an update is compatible, if the respective package is installed in both the base and the user layers.

Figure 6.13 shows the flow chart for packages added in the base layer. The first step is to determine if a corresponding entry exists in the user layer's package database. If such an entry does not exist, the base layer is compatible and the entry from the base layer's package database is copied to the resulting package database. Even in the rare case of the user having both installed the package and removed it afterwards, a masking incompatibility is impossible. In this case, the package would have been installed in the user layer, where it can be deleted. Consequently, no whiteout files would have been created that could hide the added package's files.

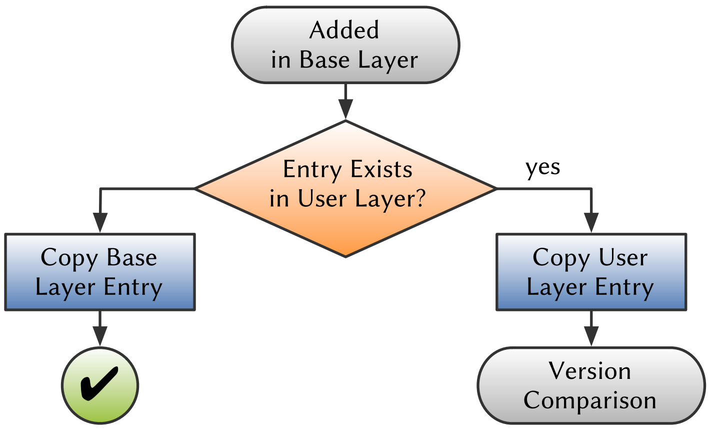


Figure 6.13 Required Checks for Added Packages. This flow chart shows a number of tests that have to be performed for added packages.

If such an entry exists in the user layer's package database, the added package is already installed in the user layer. In this case, the files added in the base layer are hidden by the ones in the user layer. Thus, the entry from the usage layer's package database is copied to the resulting package database and the version comparison described above is necessary to determine if the base layer is compatible.

Figure 6.14 shows the flow chart for packages removed in the base layer. The first step is to determine whether the entry in the user layer's package database has been 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. Therefore, the base layer is compatible and the entry from the base layer's package database is copied to the resulting package database.

However, one of the two rare cases described for updated packages can also cause a masking incompatibility for removed packages: the removal and subsequent reinstallation of the package by the user. In this case, the removed package would still be available in the composite disk image and the base layer would be incompatible. Thus, the existence of files belonging to this package needs to be checked in the same way as for updated packages.

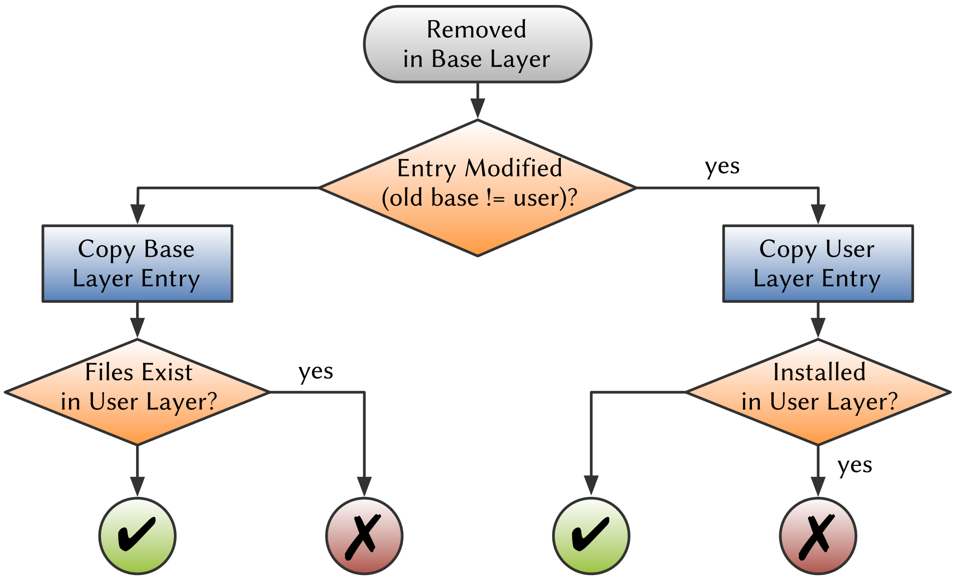


Figure 6.14 Required Checks for Removed Packages. This flow chart shows a number of tests that have to be performed for removed packages.

In case of a modified entry, the package has either been updated, downgraded or removed. In all three cases, the entry from the user layer's package database is copied to the resulting package database. If the package was removed, the removal in the base layer succeeded irrespective of the whiteout files created when the user removed the package. In this case the base layer is compatible. Otherwise the package is still available in the composite disk image and the base image is not compatible.

6.4.2.3 Detecting Relation Incompatibilities

The easiest way to detect relation incompatibilities 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 if only a few packages were modified this approach is not ideal. The current implementation uses the dpkg binary for the version comparisons required to check constraints that specify versions. A specialized Python library for version comparisons might help to improve the performance of the full constraint check. Furthermore, during the development of these tools some package databases with strange quirks were encountered, e.g., packages conflicting with themselves. The full constraint check necessarily notices these quirks and generates corresponding warnings that are typically false positives.

A more efficient approach to check for relation incompatibilities is presented below. It leverages the knowledge about all three involved package databases gained during merging. This approach consists of two steps. In the first step, only the packages that were added or updated are considered. For each of those packages, the algorithm ensures that the dependencies are satisfied and that no conflicting package is installed.

The second step considers the packages that where not changed during the update. For each of those packages, the algorithm checks whether the lists of dependencies and conflicting packages contain the names of changed packages, i.e., added, updated, or removed packages. Only the constraints that concern changed packages are verified, whereas the remaining constraints are considered as satisfied. By reducing the amount of constraints that are checked, this approach in particular reduces the number of time-consuming version comparisons.

6.4.3 Online Penetration Suite

The Online Penetration Suite is implemented in the Java programming language. Virtual machines are controlled using the Java binding of the libvirt library, whereas the Nessus vulnerability scanner is invoked using the Apache XML-RPC library and the reports of the vulnerability scanners are processed and converted using the Java API for XML Processing (JAXP).

6.4.3.1 Controlling the Vulnerability Scanners

Both OpenVAS and Nessus use a server component that is responsible for the actual scan process, storage of scan results, etc. Before a virtual machine can be scanned, these server components need to be started unless they are already running. When the server component is running, the OPS just needs to initiate and monitor the actual scan processes[24].

The adapters use different techniques to control and monitor the actual vulnerability scanners. OpenVAS provides a command line interface that expects a test configuration in the form of an XML file. The corresponding adapter creates an appropriate configuration file for the requested vulnerability scan and passes its name as an argument to the omp binary. Monitoring of OpenVAS is implemented by reading and analyzing the output of the client. Nessus, on the other hand, provides an XML-RPC API that is used to start and monitor the actual scan process. Both adapters contain code to convert the proprietary report formats into the unified OPS format that is shown in the next section.

The adapters take care of selecting sensible default values for the myriad of options both vulnerability scanners provide to adapt the scanning process. These options range from options that influence a single test to options that enable or disable entire test

[24] Optionally, the virtual machine itself needs to be started beforehand using the libvirt library.

sets. This is done to guarantee a seamless scan process that is important because the OPS is intended as an automated tool and scan processes must not fail or return false results because of missing options. The preset values the adapters use are configurable in specific configuration files.

6.4.3.2 Structure of the Reports

The summary report contains a summary of all detected vulnerabilities. An exemplary report for a virtual machine with the hostname `pentestvm0.local`[25]that has been scanned with both OpenVAS and Nessus is shown below. The virtual machine has two vulnerabilities that have been assigned a high risk factor, three vulnerabilities with a medium risk factor, and two vulnerabilities with a low risk factor. The remaining 31 vulnerabilities without an assigned risk factor are of less importance.

```xml
<summary>
  <host>
    <name>pentestvm0.local</name>
    <scanners>
      <scanner>OpenVAS</scanner>
      <scanner>Nessus</scanner>
    </scanners>
    <findings>
      <vulnerabilities>
        <risk_factor>NONE</risk_factor>
        <count>31</count>
      </vulnerabilities>
      <vulnerabilities>
        <risk_factor>LOW</risk_factor>
        <count>2</count>
      </vulnerabilities>
      <vulnerabilities>
        <risk_factor>MEDIUM</risk_factor>
        <count>3</count>
      </vulnerabilities>
      <vulnerabilities>
        <risk_factor>HIGH</risk_factor>
        <count>2</count>
      </vulnerabilities>
    </findings>
  </host>
</summary>
```

Summary Report Created by the OPS for an Exemplary Virtual Machine.

The combined report contains the reports of all enabled vulnerability scanners in the unified OPS format. One of the vulnerabilities with a high risk factor is shown in more detail in the excerpt of the combined report shown below. The report includes important information that can be used to fix the vulnerability. The structure of the combined report can also be seen in the excerpt: it is a collection of individual reports generated for a specific virtual machine using a specific vulnerability scanner. In this

[25] This machine has been started automatically by the OPS and is thus assigned a generated hostname.

example, there is one virtual machine and two scanners, therefore two reports are included in the combined report.

```
<reports>
  <report>
    <host>pentestvm0.local</host>
    <scanner>OpenVAS</scanner>
    <vulnerabilities>
      (...)
      <vulnerability>
        <title>Microsoft Outlook SMB Attachment Remote Code Execution
     Vulnerability (978212)</title>
        <port>general/tcp</port>
        <risk_factor>HIGH</risk_factor>
        <description>
          Overview: This host has critical security update missing
     according to Microsoft Bulletin MS10-045.
          (...)
          CVE : CVE-2010-0266
          BID : 41446
        </description>
      </vulnerability>
      (...)
    </vulnerabilities>
  </report>
  <report>
    <host>pentestvm0.local</host>
    <scanner>Nessus</scanner>
    <vulnerabilities>
      (...)
    </vulnerabilities>
  </report>
</reports>
```

Excerpt from the Combined Report Created by the OPS for an Exemplary Virtual Machine.

6.4.4 Security Monitoring

The various parts of the Security Monitoring System are implemented using various programming languages and libraries that optimally fit the corresponding requirements. An overview of the implementation is given below. Further details can be found in [16].

Java Applications are monitored with a set of sensors written in Java using the Java Management Extensions of the JVM. Additional information about this kind of applications is gathered using *btrace*. System behavior is monitored both using user space and kernel space sensors. The former are implemented in the Python language, whereas the latter are implemented in C. The hypervisor is monitored using both libvirt and *libvmi* [85], two generic libraries for interacting with the hypervisor, as well as the *QEMU Machine Protocol* that is specific to the QEMU/KVM hypervisor.

The communication between the sensors and the Accept VM is abstracted by a set of libraries provided for the C/C++, Python, and Java languages. For performance reasons, plain TCP connections are used instead of a message queuing system. The events are encoded using the MessagePack [57] library.

The Action Framework is implemented in the Python language. If required, integration options for interacting with the Action Framework from C/C++ or Java code exist. Actions themselves are implemented using signed Python scripts. The signatures are used to prevent attackers from abusing the Action Framework.

The ACCEPT-VM is secured using a combination of available security mechanisms: Grsecurity [110], AppArmor [14], and AIDE [5] are used to implement PaX address space protection, role-based access control, file system integrity checks, kernel auditing and executable protection. The Snort intrusion detection system is used to detect network attacks against the ACCEPT-VM. If the QEMU/KVM hypervisor is used, the processes of the virtual machines are protected using SELinux [104] in the host operating system.

6.5 Experimental Results

In this section, the four proposals that aim to increase virtual machine security are evaluated.

6.5.1 Update Checker

Two measurements have been conducted to evaluate the Update Checker. The first measurement is a local measurement that evaluates the performance of the three major components of the Update Checker: Machine Importers, Repository Importers and Scan Engine. The second measurement tests the performance of the Remote Importer.

In the first measurement, three Debian GNU/Linux 6 and three Fedora 15 virtual machines were used to evaluate the three components individually. The virtual machines have different numbers of installed packages to test how the components deal with increasing package database sizes. The number of installed packages, enabled repositories, and the average import and scan times out of 20 measurements are shown in Table 6.4.

Table 6.4 Individual Component Benchmark. The table shows the result of the evaluation of all Update Checker components using six different virtual machines.

				Average Execution Time (s)		
ID	Distribution	Number of Packages	Number of Repositories	Machine Importer	Repository Importer	Scan Engine
D1	Debian	563	2	0.04	2.39	0.44
D2	Debian	867	2	0.06	2.80	0.44
D3	Debian	1,493	2	0.07	2.68	0.78
F1	Fedora	591	2	0.03	13.59	0.38
F2	Fedora	1063	3	0.04	14.84	1.00
F3	Fedora	2,283	4	0.05	15.38	2.10

In the first part of this measurement, the different machine importers were tested. All required files, i.e., the package database and the repository configuration, are copied to the host the Update Checker is running on prior to the evaluation, thus network communication is not part of the measurement. In the case of Fedora and `rpm`, the `rpm --query --all --queryformat '%{nevra}\n'` command is executed on the source machine to extract the list of installed packages including version identifiers from the package database and the list is copied to the host instead of the package database itself. This workaround is required to deal with software incompatibilities[26]. Since the list contains only the information the Update Checker requires, this approach gives the `rpm` Machine Importer an advantage over the `dpkg` Machine Importer that has to process the entire package database including unneeded information for the Debian virtual machines. The results confirm this assumption.

The second part of the measurement evaluates the time required to download and parse the repository databases of all repositories enabled for the virtual machine without using the repository cache. The times measured are thus artificial and are only of little relevance for actual usage of the Update Checker, but can be used for evaluating the Repository Importers. The time required for importing the Debian repositories machines is quite stable. For the Fedora repositories, however, the import time increases, certainly because of the increased number of enabled repositories. The very bad performance of the Fedora Repository Importer in general is caused by the use of an XML format for storing the repository database.

In the last part of the measurement, the algorithm that actually checks for updates is evaluated. Again, the growing number of packages causes the increase of the execution times. The reason for the worse results for Fedora are probably the longer and more complex version numbers that can be found in Fedora, making the comparison harder and more time-consuming.

Except for the time required by the Repository Importer for Fedora machines, the measured values are promising. The relatively long time required for importing yum repositories is compensated by the repository cache. It ensures that every repository is downloaded and parsed only once during the its configurable validity period. In all measurements above, each virtual machine was evaluated individually, i.e., all caches were cleaned after every measurement. To evaluate the influence of the repository cache, another measurement has been conducted that represents a more realistic scenario. The same six machines used in the last measurement were scanned at once, whereby the repository cache was active. The experiment was repeated 20 times and the average times are shown in Figure 6.15a. The results show that the repository cache is very effective in cutting down the time required to scan multiple virtual machines for updates.

To evaluate the scalability of the Update Checker (and applicability for physical machines), 115 physical nodes that are part of the MaRC cluster were imported into the Update Checker. All machines were scanned at once using the repository cache. The experiment was repeated 20 times and the time required to scan all virtual machines was calculated. The results shown in Figure 6.15b provide evidence for the scalability

[26] The version of `rpm` in Debian GNU/Linux 6 was not able to read the package database of Fedora 15.

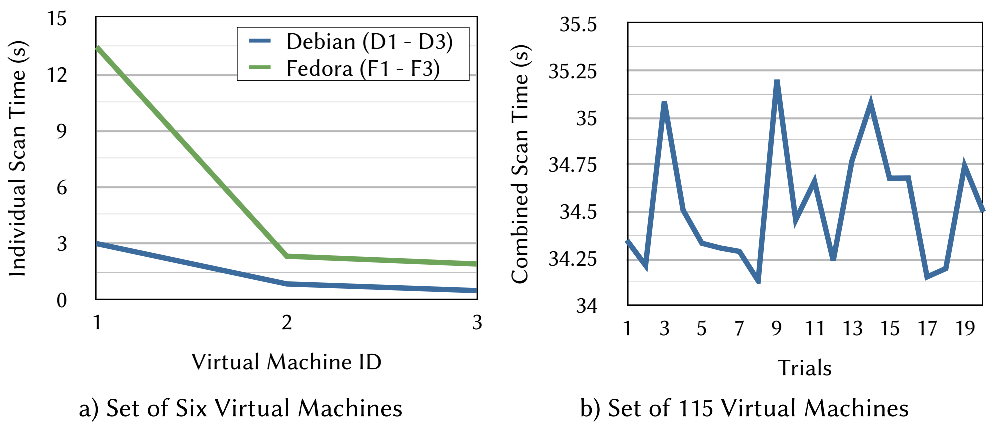


Figure 6.15 Benchmark of Complete Scan Processes. Chart a) shows the time required to scan each of the six virtual machines *D1 - D3* and *F1 - F3* (see Table 6.4) for updates with the Repository Cache enabled. The virtual machines are already imported at the beginning of the measurement, but the repository cache is empty. Chart b) shows the combined time required for scanning a set of 115 Debian GNU/Linux 5 machines for updates with the Repository Cache enabled. The virtual machines are already imported at the beginning of the measurement and the Repository Cache is primed.

of the Update Checker. The average scan time was 34.53 seconds for all 115 machines or 0.30 seconds per machine.

The second measurement was conducted to measure the import time of the virtual machines using the Remote Importer. This involves gathering all required files, executing `rpm --query --all --queryformat '%{nevra}\n'` in the case of rpm based distributions, sending either the package database or the extracted package list together with the repository configuration to the Update Checker and the actual import process. Each virtual machine was imported 10 imported 10 times. The results are shown in Figure 6.16.

As expected, the duration of the import process grows with the number of installed packages. Generally, the import process is faster for dpkg-based virtual machines than for rpm-based virtual machines. The reason is certainly the use of the rpm binary to extract the information from the package database. This step was not considered in the first measurement.

6.5.2 Centralized Update Process

To evaluate the centralized update process the Virtualized Grid Computing use case is revisited (see Section 4.5.3). The update of the base layer is comprised of three updated and one added packages with a total disk usage of about 12 MiB[27]. Additionally, the update layer contains about 40 MiB of dpkg metadata, e.g., package lists, the compressed packages and the updated package database. Table 6.5 lists image file sizes

[27] In fact, this is actually the update that fixes the OpenSSH bug already mentioned in Section 6.3.2.4.

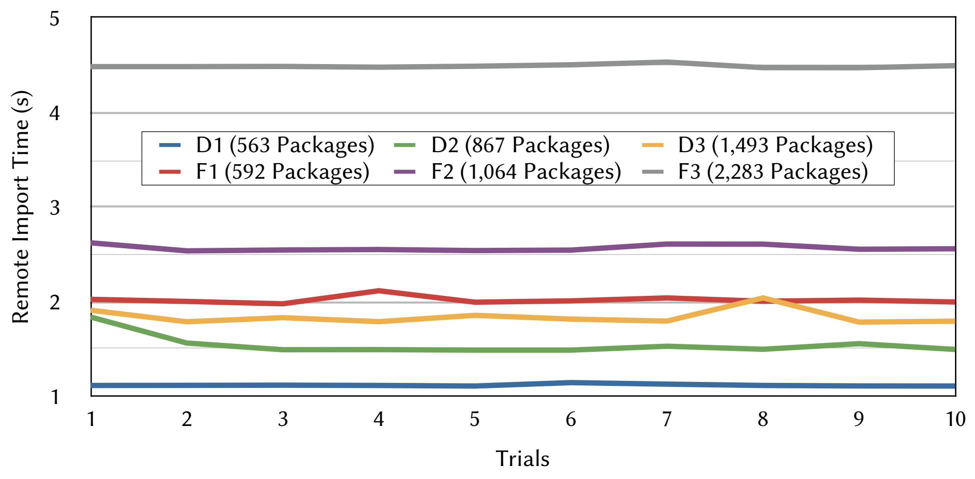


Figure 6.16 Benchmark of the Remote Importer. This figure shows the time required to import the six virtual machines *D1 - D3* and *F1 - F3* using the Remote Importer.

and transfer times for the regular and layered virtual machine as well as the update layer. It is a reproduction of Table 4.12 that is extended with the numbers for the update layer (highlighted). The meaning of the different transfer times is explained in the description of the use case.

Table 6.5 Size and Transfer Time in a Single and a Multi Site Scenario. Transfer times of virtual machine, both regular and layered, and an update layer between compute nodes of a cluster and between two clusters at different Grid sites, optionally compressed. Extended reproduction of Table 4.12, the added line is highlighted.

		Transfer Time (s)		
		Single Site	Multi Site	
Image	Size (MiB)	Uncompressed	Uncompressed	Compressed
Regular Image	691.1	40.6	660.8	460.1
Base Layer	666.4	39.1	636.9	443.5
User Layer	67.7	14.4	101.5	91.6
Update Layer	72.3	15.1	106.2	100.1

Without update layers, every time the user updates his virtual machine and submits a job to the Grid, the entire disk image of the virtual machine or the updated base layer has to be copied to the particular execution hosts, because updating a virtual machine invalidates all copies of the old version in images caches of execution hosts. In this example, around 690 MiB or 666.4 MiB need to be copied to the execution hosts[28]. With update layers, this is reduced to 72.3 MiB. The transfer time is reduced from 660.8 seconds or 636.9 seconds to 106.2 seconds in the worst case. Applying the changes in the update layer to the base layer takes additional 4.05 seconds for this

[28] Perhaps even a little bit more, because of the additional package. The updated regular virtual machine image and the updated base layer were unfortunately not included in the transfer measurements.

update. Nevertheless, this is still much faster compared to copying the entire image files.

When the layered virtual machine is started with the updated base layer for the first time, the package databases needs to be merged and the check for incompatibilities needs to be done. For this virtual machine, both steps took 0.36 seconds in total. To evaluate the scalability of the merge algorithm, a second measurement was conducted with a user layer containing 1231 packages instead of 14 and the same base layer and update layer. The time needed to merge the package database and check for incompatibilities grew to 1.01 seconds in this case (all values are the means of 60 measurements).

6.5.3 Online Penetration Suite

The following section presents measurements related to the OPS. All tested systems are Xen domU virtual machines running Debian GNU/Linux 6 (Squeeze) executed on Intel Pentium 550 machines running at 3.4 GHz with 1 GiB of memory. The OPS node is an Intel Xeon E5520 machine running at 2.26 GHz with 1 GiB of memory. All systems are interconnected with switched fast Ethernet.

The first measurement evaluates the total runtime of the OPS depending on the number of scanned virtual machines. The results of the measurement are shown in Figure 6.17. The OPS used both vulnerability scanners in parallel while the number of scanned virtual machines was increased with every run. To get a robust mean, 100 trials were performed. Scanning took 684 seconds, 859 seconds, 1,056 seconds, and 1,279 seconds on average for one, two, three, and four scanned virtual machines. The scan time can probably be reduced by adapting the configuration of the vulnerability scanners to the scanned virtual machines, i.e., disable scanning for Windows vulnerabilities in Linux virtual machines.

The runtime obviously increases linearly with the number of tested systems and it is more efficient to scan multiple virtual machines in parallel instead of scanning them sequentially. This results confirms the conclusion that scanning virtual machines as part of continuous maintenance operations is more sensible than scanning them shortly before deployment, because the former approach allows to scan them in batches of multiple virtual machines.

In order to evaluate the effectiveness of the OPS, multiple scans of virtual machines running different versions of Debian GNU/Linux were conducted. The unpatched release versions of Debian GNU/Linux 4 (Etch, released April 2007), 5 (Lenny, released February 2009), 6 (Squeeze, released February 2011) and 7 (Wheezy, released May 2013) were used for this test. The results are shown in Table 6.6.

The OPS successfully revealed a number of security vulnerabilities in all tested versions, including two high-risk vulnerabilities that are present in all versions. After a detailed examination of the results, the high-risk vulnerabilities are false-positives[29].

[29] A Windows vulnerability and a skipped check that detects a vulnerability in the network switch a host is connected two.

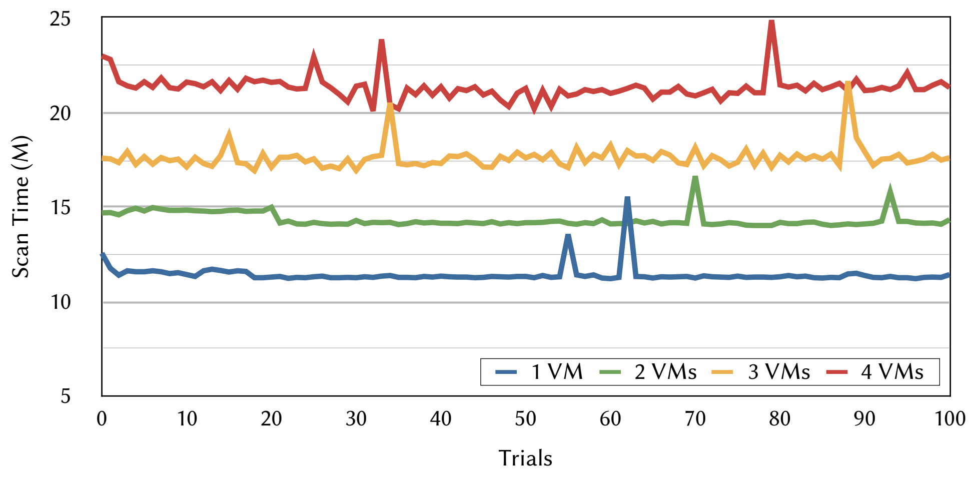


Figure 6.17 OPS Scan Times. This figure shows the time the OPS needs for scanning one to four virtual machines for vulnerabilities.

Table 6.6 OPS Scan Results for Various Debian Versions. Number of security vulnerabilities the OPS detected in different versions of Debian GNU/Linux.

	Risk Level			
Operating System Version	*None*	*Low*	*Medium*	*High*
Debian GNU/Linux 4 (Etch)	14	2	0	2
Debian GNU/Linux 5 (Lenny)	43	2	3	2
Debian GNU/Linux 6 (Squeeze)	44	2	3	2
Debian GNU/Linux 7 (Wheezy)	43	2	3	2

The medium-risk vulnerabilities are an activated *Zeroconf* daemon (*avahi*) and two problems in the TCP stack: implementation of TCP timestamps[30] and a *possible* predictability of TCP/IP *Initial Sequence Numbers*. The oldest release is not affected by these vulnerabilities because the avahi daemon was not activated by default and an older kernel version is used. The low-risk vulnerabilities are the answering of ICMP timestamp requests[31] and support for password authentication for SSH connections.

6.5.4 Security Monitoring

In this section, a performance evaluation for the entire monitoring system is presented: a measurement of the round trip time, i.e., the time between the generation of an event by a sensor and the execution of a resulting action. Further measurements, e.g., sensor overhead, performance of the CEP engine and Event Store, can be found in [16].

To demonstrate that the proposed monitoring system can be used in a real world setting, the difference between the creation time of an event created by a sensor and

[30] This might allow an attacker to compute the uptime of the system under some circumstances. Using this information, he might be able to draw conclusions about the patch level of the system.

[31] This enables an attacker to learn the system time that may help him to defeat time-based authentication protocols.

the execution time of an action triggered by that event is calculated. This is regarded as round trip time through the entire monitoring system.

A sensor explicitly designed for this benchmark sends an event consisting of an increasing numeric identifier and the current time of the guest machine. The experiment is based on a simple select-all EPA to redirect this event to the Action Framework. An action that compares the current time to the creation time stored in the event is selected by the Action Framework and sent to a user space actor for execution. There, the time values are then compared and the resulting round trip time is calculated. The actor is placed in the same virtual machine as the sensor to avoid inaccuracies caused by imperfect clock synchronization.

Figure 6.18 shows the round trip time measured if the sensor's event generation rate is limited to a single event per second. In this idle load situation, the system has an average reaction time of 0.007 seconds.

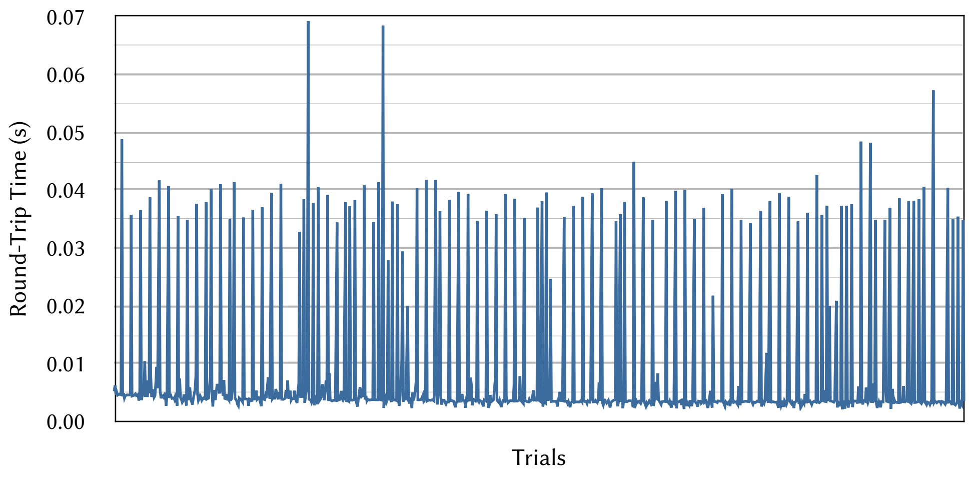


Figure 6.18 Round-Trip Time (Limited Event Rate). This chart shows the round trip time through the entire monitoring system under idle load. (Source: [16])

Figure 6.19 shows the round trip time measured if the sensor's event generation rate is unlimited. This results in about 23,700 actions triggered per second. Since this experiment investigates the monitoring system at the boundaries of the virtual hardware with a triggering rate of over 23,000 actions per second, a mean value of 1.82 seconds for the round trip time is still within an acceptable timeframe and indicates how well the monitoring system operates under heavy load.

6.6 Summary

In this chapter, four proposals have been introduced that work hand in hand to improve the security of virtual machines in the continuous maintenance, deployment, and execution phases of their lifecycle. These phases are the most important ones when with regard to security and thus the proposals help to increase the overall security of Virtualized Grid and Cloud Computing environments.

The first proposal is the Update Checker that is able to scans virtual machines for

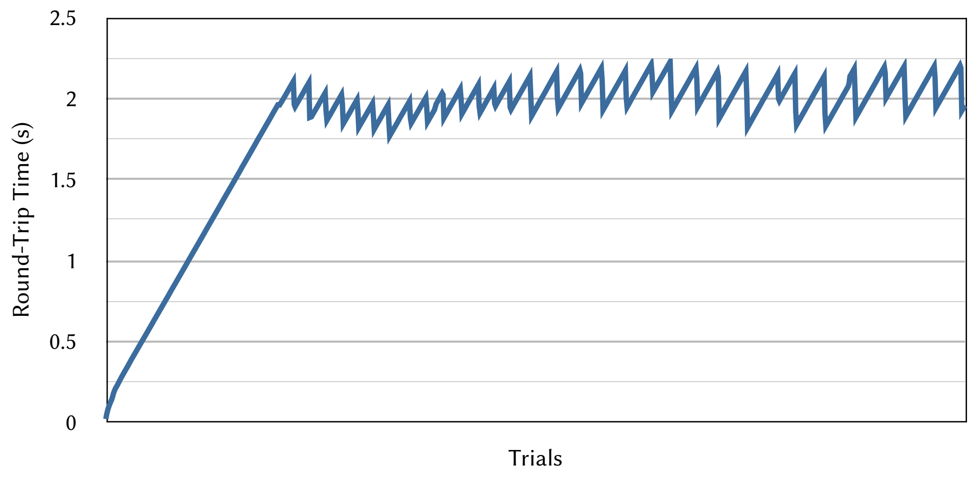


Figure 6.19 Round-Trip Time (Unlimited Event Rate). This chart shows the round trip time through the entire monitoring system under heavy load. (Source: [16])

outdated software in an efficient manner, which is a challenging task especially for dormant virtual machines. It relies on a central database containing the information from the package databases of the individual virtual machines. Using this database, it is able to perform its scan even for dormant virtual machines.

The second proposal is a concept for centrally updating virtual machines using the image composition technique presented in Chapter 4. This proposal solves the scalability problem of installing updates for large numbers of virtual machines by limiting the installation to a single or a few base layers. Although this proposal has some limitations, the concept and the accompanying tools are a valuable addition to an administrator's toolset for cases where a large number of virtual machines need to be updated in a very short timeframe.

The third proposal is the Online Penetration Suite that integrates vulnerability scanners into the lifecycle of virtual machines either as part of the continuous maintenance or as interim stage in the deployment process. With the help of the vulnerability scanners, misconfigurations or insecure services can be detected in virtual machines before an attacker is able to exploit them.

The last proposal in this chapter is the concept of a security monitoring system for virtual machines that also covers the execution phase. The system is able to detect, analyze, and handle security anomalies including both known and yet unknown security vulnerabilities by using a system of sensors at the hypervisor, guest OS, and application level, CEP technology for detecting anomalies, and a flexible framework for responding to those anomalies.

All four proposals have demonstrated satisfying performance in a variety of measurements.

*"All fixed set patterns are incapable of adaptability or pliability.
The truth is outside of all fixed patterns."*

Bruce Lee (1940–1973)

7 Virtual Machine Migration

7.1 Introduction

Migration of virtual machines is an enabling technology in Virtualized Grid and Cloud Computing environments. Its main purpose is load balancing to optimize resource usage with regard to either performance of the virtual machines or energy conservation. Virtual machine performance is degraded if either the number of virtual machines on a single execution host is too high or the combined CPU, memory and I/O utilization is close to the capacity of the execution host. In such a case, some of the virtual machines can be moved to a less loaded execution host. Contrary, it is possible to conserve energy by consolidating virtual machines from multiple slightly loaded execution hosts on a smaller number of (higher loaded) executions hosts and turning of unused execution hosts [62].

The migration of virtual machines is also useful for planned maintenance of execution hosts in the presence of long-running virtual machines that cannot be shut down. Additionally, the migration of virtual machines is advantageous in case of hardware problems. In both cases, it allows to migrate the virtual machine to another execution host and carry out the maintenance or repair operations on the original execution host.

The challenge with virtual machine migration is the desire to do *live migrations*, i.e., migrating a virtual machine while it is running without any downtimes visible to the users. This is achieved by keeping the network connections open during the migration process. The live migration is a completely transparent process from the user's perspective. Live migration poses additional challenges when the virtual machines use local storage for their associated virtual disks instead of shared storage: the complete disk state needs to be transferred to the destination host to prevent residual dependencies on the source host. This is complicated by the fact that the virtual machines keep running and are hence altering the disk state during the transfer process. In the remainder of this chapter, the local storage case is studied.

In the targeted Virtualized Grid and Cloud Computing environments, each virtual machine is equipped with an *ephemeral* disk that loses its content after shutdown. Important data is therefore not stored inside the virtual machine, but on external storage systems. This peculiarity facilitates using a composite disk image as virtual disk: a composition of the actual virtual machine image and a temporary layer. The use of a composite disk image guarantees that the virtual machine image is never modified during the runtime of the virtual machine, because all modifications to the virtual disk are redirected to the temporary layer. With regard to live migration, this is a significant advantage: the migration process only needs to take care of transferring the temporary layer. The actual virtual machine image can be transferred to the destination host in advance[1].

In this chapter, a concept for efficient synchronization of virtual disks during live migrations is presented. The first technology this concept is based upon is the image composition proposed in Chapter 4. Three different configurations for composite disk images are presented and assessed with regard to efficient live migration: a memory-only, a disk-based, and a hybrid configuration.

Most hypervisors are not implemented with local storage of disk images in mind when it comes to live migration. Consequently, they do not take care of transferring the virtual machines' disks between execution hosts, but instead rely on a shared storage solution that is accessible by both execution hosts. The second technology the proposed concept is based on is a disk synchronization mechanism originally designed for high availability clusters. It ensures seamless and transparent synchronization of disk images prior to and during the migration of a virtual machine.

The presented concept is implemented in the XGE [147, 138] that provides a basic infrastructure for managing a cluster of virtual machines: machines can be started and stopped per job, and user specific disk images can be deployed to the corresponding execution hosts before the machines are started. The XGE offers a live migration facility to migrate virtual machines between execution hosts during runtime. The novel concept of virtual machine image synchronization introduced in this chapter enhances the current migration mechanism and allows cheaper and faster virtual machine migrations to ease load balancing.

The live migration of a virtual machine obviously deals with running virtual machines, and thus is focusing the execution phase in the lifecycle of a virtual machine. Furthermore, the live migration of a virtual machine can be seen as the deployment of a running virtual machine on another host, whereby virtual machines are normally deployed when they are not running. This particularly applies to the migration concept based on image composition: the virtual machine image does not have to be transferred to the target host from the source host of the migration, but can be deployed using the regular means of image deployment. Thus, the concept proposed in this chapter also affects the deployment phase in the lifecycle. The phases in the

[1] Actually, it might not even be required to transfer the virtual machine at all, if it is already cached in an Image Cache at the destination host.

lifecycle of a virtual machine that are related to using the proposed migration concept are shown in Figure 7.1.

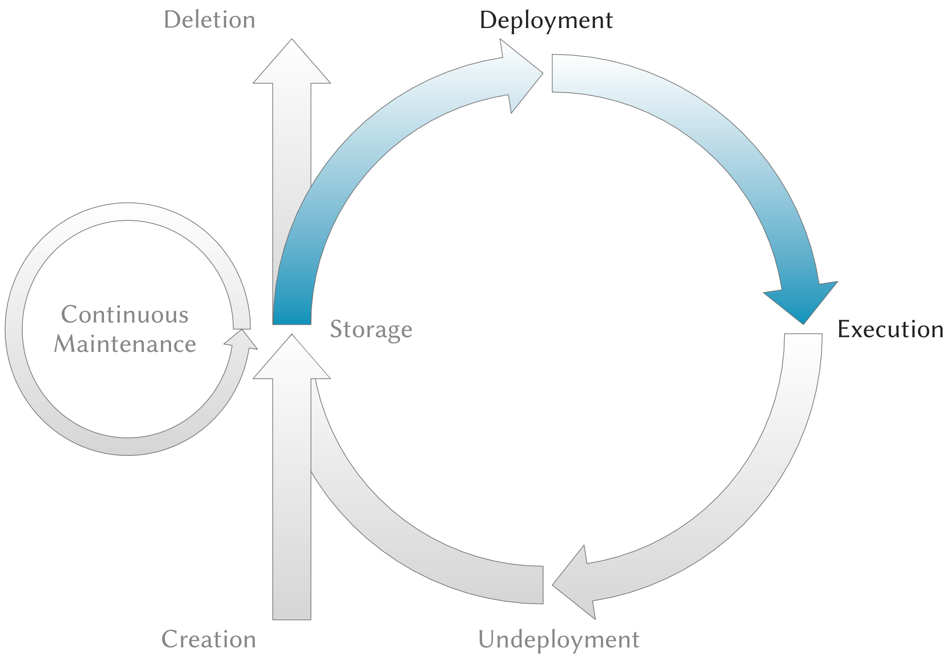


Figure 7.1 Related Virtual Machine Lifecycle Phases. The primary focus of virtual machine migration regarding the lifecycle of a virtual machine is obviously the execution phase. Since the proposed migration approach can use the regular deployment mechanism to transfer read-only parts to the destination node, the deployment phase is affected as well.

Parts of this chapter have been published in [65, 162].

7.2 Related Work

7.2.1 Virtual Machine Migration

One of the first works considering migration of virtual machines has been published by Kozuch et al. [81]. The basic motivation of the proposal is mobility of users: a virtual machine is suspended on the source hardware and resumed on the destination host. Local state, memory pages and persistent storage are copied using distributed file systems like NFS or Coda [134]. While this approach targets users who want to continue their work on different locations without the need for carrying the physical hardware, e.g., a laptop, with them, it does not fit into a high-performance computing environment where the migration should have a minimal impact on the migrated virtual machine. Nevertheless, a slightly modified version of the approach that copies the persistent storage directly to the target machine instead of using shared storage would be a very simple and efficient solution for scenarios tolerating longer downtimes of virtual machines, e.g., batch jobs running in virtual machines.

7.2.2 Live Migration in Local Networks

Fundamental research on virtual machine live migration was done by Clark et al. [28]. Their approach is based on the observation that write accesses to main memory are concentrated to a small number of pages (the ‘writable working set’) for a certain time period. Hence, they pre-copy the memory pages to the destination host while the virtual machine keeps running on the source host. Dirty memory pages are re-transferred in several iterations until the remaining set of dirty pages is small enough or a maximum number of iterations is reached. Then, the virtual machine is suspended, all dirty pages and the internal state are copied, and the virtual machine is resumed on the target host. This solution has been designed for live migration in local networks assuming shared storage between the virtual machines. Xen has adopted this approach.

An alternative approach is taken by Hines et al. [66]: instead of pre-copying the memory pages before migration, only the internal state is transmitted, and the virtual machine is resumed immediately at the destination. Memory pages are copied on demand as well as by prepaging techniques, reducing the number of costly page faults over the network. The authors showed that the post-copy approach can improve some metrics, such as the total number of transferred pages, the total migration time and the network overhead compared to the pre-copy approach. An interesting mechanism is the ‘dynamic self-ballooning’ method introduced in their work: unused memory pages can be reclaimed from the virtual machine, which reduces the total number of pages that must be copied during a migration process and thus yielding a migration speed up.

Voorsluys et al. [163] have extensively analyzed and evaluated the cost of virtual machine live migration in a Cloud environment. A number of case studies with representative workloads were executed to measure the performance impact on the applications running inside a virtual machine during live migration. The results show that the impact of live migration on a heavy loaded virtual machine cannot be neglected especially when service level agreements need to be met. Live migration of virtual machines with a slightly reduced load is relatively uncritical, and the most stringent service level agreements (99th percentile) can still be met.

7.2.3 Storage Migration

Bradford et al. [22] and Luo et al. [92] propose mechanisms for entire system migration including local persistent state. Both use special disk drivers intercepting disk writes and reads to make the transfer of the local state possible during live migration.

Bradford et al. [22] use a two-phase transfer: first, the complete disk image is copied from source to destination while all new write operations are recorded and sent as ‘deltas’ to the destination in parallel. The second phase consists of migrating the main memory and the CPU state via the Xen migration mechanism and to apply all deltas to the disk image on the destination. The virtual machine is resumed once Xen migration has finished, but all I/O operations must be blocked until the application of the deltas is complete. The authors showed that in the majority of cases the application of deltas is finished before the Xen migration and hence no additional downtime is observed. Nevertheless, there are workloads (especially write intensive workloads) that can cause a significant increase of downtime. In addition, write intensive workloads need to be throttled during migration if the write rate exceeds the network bandwidth, which would otherwise prevent the migration process from progressing.

Luo et al. [92] propose a three-phase migration approach. In the pre-copy phase, the disk image is copied incrementally to the destination host (similar to the mechanism used for transferring the memory pages during Xen live migration). Dirty blocks are tracked using a block bitmap and retransferred in subsequent iterations. In the next phase, the virtual machine is suspended, the block bitmap is copied to the destination and the virtual machine is resumed immediately without transferring the remaining dirty blocks. In the third phase - the post-copy phase - a special disk driver intercepts all disk accesses and copies the missing blocks from the source host on demand. The main drawback of this solution is its dependence on the source host. If the source host (or the network connection between source and destination) crashes before all remaining dirty blocks are transferred, the virtual machine cannot continue to run since the disk image is not consistent. Such a situation cannot occur in Bradford et al.'s approach [22].

Sapuntzakis et al. [132] present a whole-machine migration process including local storage. They use so-called 'capsules' that encapsulate a machine's state and can be moved between different machines. The main memory and CPU state is transferred offline (the machine is suspended, the state is copied to the destination and the machine is resumed), which takes several minutes even over a fast link and thus renders this approach useless for live migration scenarios in Cloud environments. Disk images are copied only on demand, reducing the total migration time at the price of a possibly infinite dependence on the source host.

7.2.4 Image Deployment

Most research on disk image deployment was conducted in the context of virtual machine migration. During the migration process, the deployment usually consists of a one-to-one relation: the disk images need to be transferred from the source to the destination. However, this area represents only a small fraction of use cases for image deployment. Schmidt et al. [138] analyzed and compared different approaches for distributing images to a large number of destinations in a Cloud environment.

7.3 Design

Both Virtualized Grid and Cloud Computing environments are very dynamic: virtual machines are constantly started on execution hosts and stopped again. In such an environment, the possibility of migrating virtual machine between differed execution hosts is an essential feature: it enables dynamic load balancing, energy efficient machine utilization, and ease of maintenance. An important property of the employed migration mechanism is transparency: neither should the user notice that his virtual machine is being migrated, nor should the virtual machine's operation be interrupted during the migration process.

Most live migration implementations, e.g., the Xen migration facility, do not handle the transfer of virtual disks between the source and target host of the migration. Consequently, virtual disks are assumed to be located on a shared medium that can be accessed from both the source and the destination host involved in a live migration. This approach leads to several problems, especially in Virtualized Grid and Cloud Computing environments:

- Accessing virtual disks via network always introduces a performance penalty compared to accessing virtual disks stored locally.
- Each virtual machine depends on the shared storage facility and a operational network connection to work properly.
- Each instance of a virtual machine needs its own working copy of an image file due to local modifications, although in a Cloud environment many virtual machines might share the same image file.

A solution that is better suited to the requirements of a Virtualized Grid or Cloud Computing environment should allow a virtual machine to access its virtual disks locally during normal operation. With regard to the live migration, this implies the need for storage migration in addition to the migration of main memory and CPU state provided by the hypervisor. The storage migration has to be implemented in a manner that it only affects the virtual machine's performance during a migration process in a negligible way.

7.3.1 Virtual Machine Image Composition

As already stated, all data written to the ephemeral disk of a virtual machine in a Virtualized Grid or Cloud Computing environment is lost after the virtual machine is shut down. Thus, the virtual disk can divided into two parts: the actual virtual machine image that contains the operating system and applications and the writable part that contains all data written during the virtual machine's runtime. If the virtual machine image is never modified, only the writable part is subject to the storage migration, whereas the virtual machine image can be copied to the target host of a live migration using regular means.

The virtual machine image composition approach proposed in Chapter 4 facilitates dividing a virtual disk in exactly the way described above. A virtual machine image is used as read-only layer in the composition, whereas a writable temporary layer is used to capture all modifications to the virtual disk. There are three potential configurations for composite disk images that are reasonable with regard to the live migration:

Memory-only Configuration — In the first configuration shown in Figure 7.2a, the virtual disk is composed of the virtual machine image and a RAM disk used as writable layer. The main advantage of this configuration is that the setup is transparent to live migration: since all modifications to the virtual disk reside in main memory, they are transferred to the target host by the normal memory copy process without requiring any special measures. A positive side effect from the virtual machine's point of view is the increased I/O performance for disk accesses. For this setup to work properly, a sufficiently large amount of RAM needs to be allocated to the virtual machine. However, there are two kinds of workloads that are not well suited for this approach: memory intensive workloads that do not leave memory for the writable layer of the virtual disk and workloads writing much data to the virtual disk.

Disk-based Configuration — In the second configuration shown in Figure 7.2b, the virtual disk is composed of the virtual machine image and a temporary disk image used as writable layer. It is better suited to workloads that write lots of data to the virtual disk. Using this configuration, the disk writes do not clutter up the main memory.

Hybrid Configuration — In the last configuration shown in Figure 7.2c, the virtual disk is again composed of the virtual machine image and a RAM disk used as writable layer. Contrary to the memory-only configuration, the RAM disk is much bigger and a swap partition residing in a temporary disk image is used as backing memory. This has the advantage of supporting memory intensive as well as disk write intensive workloads. All data written to the writable layer and not accessed again for some time will be swapped out to the temporary disk image and thus the main memory does not get filled up with unused file system content.

The challenge with the last two configurations is that they are not transparent to live migration. Most hypervisor implementations do not support live migration of virtual machines with local persistent storage, but instead rely on a shared storage system, e.g., Network File System (NFS), internet Small Computer System Interface (iSCSI) or ATA over Ethernet (AoE). A virtual disk must be accessible from both the source and target host to be able to migrate a virtual machine. However, for performance reasons it is desirable that a virtual machine has local access to its disk images. Consequently, a mechanism is needed for seamlessly transferring the disk images to a new host.

7.3.2 Disk Image Synchronization

Using local persistent storage poses some challenges when performing live migration of virtual machines. The main problem is to transfer a consistent state of the virtual disk to the destination host while the virtual machine keeps running and thus is altering the virtual disk's state. Therefore, the task is divided into two parts: copying the data and tracking changes to the data already copied (and somehow send them to the destination host).

The synchronization mechanism proposed in this paper works as follows: at the beginning of the migration process, the source host starts to copy the disk image to the destination. In parallel, all subsequent disk writes from the virtual machine to be migrated are trapped and issued synchronously to the local and the remote disk image. Synchronously means that the acknowledgment of the disk write is delayed until the

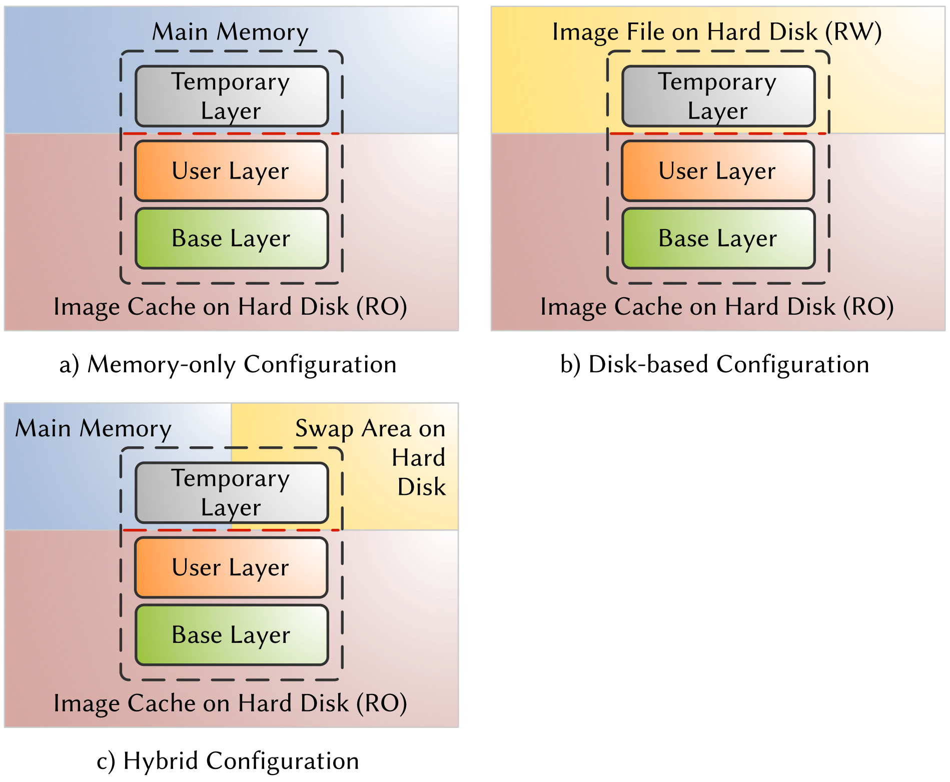


Figure 7.2 Composite Disk Image Configurations. This figure shows three different configurations for the composition of virtual disks that are reasonable with regard to live migration of virtual machines.

remote host has confirmed it. Once the background copy operation is finished, the normal live migration process is started, and during the entire live migration process, the two disk images operate in the synchronized mode. After the virtual machine is resumed on the destination host, the disk images are decoupled, and no dependence on the source host remains.

To perform the actual synchronization, the *DRBD* [88] kernel module is used; it is integrated into the mainline Linux kernel since release 2.6.33. DRBD is designed for high availability clusters mirroring a disk from the primary host to a secondary backup host and thus acting as a network based RAID-1. Figure 7.3 shows the design of the module. It presents itself to the kernel as a disk driver and thus allows a maximum of flexibility: it does neither pose restrictions on the file system used above nor the underlying disk driver managing the actual disk accesses. And it is transparent to the kernel block device facilities, which means that buffering and disk scheduling are left to the kernel as usual. The module can operate in two modes: standalone and synchronized. In standalone mode, all disk accesses are simply passed to the underlying disk driver. In synchronized mode, disk writes are both passed to the underlying disk driver and sent to the backup machine via a TCP connection. Disk reads are served locally.

Although designed for a high availability setup where disks are mirrored across the

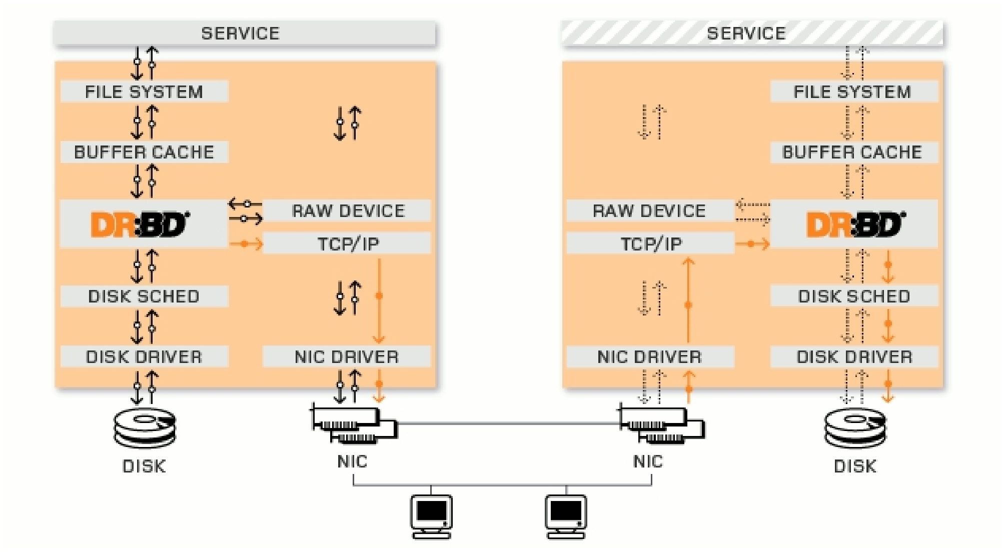


Figure 7.3 DRBD Module Overview. This figure shows an overview of the DRBD module in synchronized mode. All writes operations on the disk of the primary server (left) are send to the secondary server (right) and executed on its disk as well. (Source: [88])

network during normal operation, the DRBD module can be integrated into the live migration process for migrating local persistent storage. The source host takes the role of the primary server, and the destination host takes the role of the secondary server. During normal operation, the source host runs in standalone mode and thus writes are performed only on its local disk without any dependence on other hosts. During live migration, the DRBD driver is put into synchronized mode, which causes all disk writes to be performed synchronously on both hosts while the entire disk is synchronized in the background. Once the migration is finished and the virtual machine is resumed on the destination host, the DRBD driver on the destination host is put into standalone mode, and the source host is disconnected, removing all dependencies between the two execution hosts.

This concept has several advantages. First, there is nearly no *performance overhead* during normal operation of a virtual machine, because all disk writes are performed locally in the standalone mode of the DRBD driver. Second, the solution is *reliable*: if a migration fails, the virtual machine can keep running on the source host, and due to the synchronous writes on both hosts, the virtual machine has a consistent disk state on the destination host after a successful migration. Furthermore, there are *no residual dependencies*: once the virtual machine is resumed on the destination host, no dependency on the source hosts remains. In particular, no disk writes are ever issued on an inconsistent disk (such as, for example, in the approach of Luo et al. [92]). Finally, the synchronization of virtual disks is completely *transparent* to the running virtual machine and the mechanism is *hypervisor-independent* and can thus be used with any hypervisor that supports live migration.

The total migration time is increased compared to a memory-only migration, whereby the additional time grows linearly with the size of the disk image that needs to be transferred to the destination host. Fortunately, the use of virtual machine image composition reduces the amount of data to be synchronized by separating read-only from writable parts of the virtual disk. The virtual machine image – the read-only part – can be fetched from a separate image pool in parallel or might even be already cached at the destination host so that no extra copy process is needed. Furthermore, DRBD allows for checksum-based synchronization, i.e., only blocks that differ between source and destination are transferred. If sparse image files are used on both sides, all unused blocks are implicitly zero-filled and are thus identical on both sides, reducing the total amount of data to copy.

Background disk synchronization and the transfer of the main memory are performed sequentially, so that they do not affect each other in a counterproductive way: since both tasks generate network traffic, the memory dirtying rate would exceed the transfer rate (due to the parallel disk synchronization) much faster, resulting in the abort of the iterative copy phase and thus a longer downtime of the virtual machine. Furthermore, disk synchronization usually takes much longer than the memory copy process and thus an exact timing would be difficult.

The amount of bandwidth consumed by background synchronization can be dynamically configured in the DRBD driver. This enables the administrator to find an appropriate trade-off between total migration time and performance degradation of the virtual machine due to high network consumption. Write intensive workloads are implicitly throttled by the synchronous nature of the disk writes, such that the disk write rate never exceeds the network transfer rate, which would render any disk synchronization mechanism useless. Bradford et al. [22] employ explicit write throttling whenever the write rate exceeds a predefined threshold. Luo et al. [92] stop their pre-copy phase proactively if the disk dirty rate is higher than the transfer rate, resulting in a much longer post-copy phase where the destination host still depends on the source host and the virtual machine runs with decreased performance.

No additional downtime of the virtual machine is introduced, because the virtual machine can be resumed without any further delay once the live migration process of the hypervisor has finished. In contrast, the approach of Bradford et al. [22] delays all disk I/O of the virtual machine until the remaining writes are applied to the disk on the destination host. This can cause an additional delay for write-intensive workloads.

7.4 Implementation

The synchronization mechanism using DRBD devices has been implemented in the XGE, an open source Virtual Machine Manager. The XGE is written in the Python programming language, and thus the controller that handles the synchronization mechanism has also been written in Python. The current implementation works with Xen as the back end hypervisor, although most of the code is not depending on Xen.

7.4.1 DRBD Device Configuration

A two-node setup consists of a pair of DRBD devices that are identified by their path in the file system. Additionally, the DRBD endpoints communicate via two separate TCP connections, and thus they have to agree on port numbers for both sides. Consequently, a hostname, port number and path identify a DRBD endpoint. To reduce the global configuration overhead, the nodes manage their resources locally (see next section). The actual device names are abstracted by symbolic links.

In this context, DRBD devices work in different modes throughout their lifetime: usually, a DRBD device runs in standalone mode as a pure bypass to the backing block device. In this mode, disk I/O performs nearly with native speed. In the pre-migration phase, the DRBD device on the source host is connected to the endpoint on the destination host, and the pair of devices runs in primary/secondary mode during the initial synchronization, whereby only the source node is allowed to write to the device. Just before the actual migration starts, the devices are put into primary/primary mode. This is necessary because Xen checks for write access on all associated block devices before initiating a live migration. From the devices' point of view, this mode allows both ends to issue write requests simultaneously. However, this will never happen due to the nature of a virtual machine migration: a virtual machine is always running on a single host and thus will always access the device only through one endpoint at a time.

7.4.2 Node Setup

Apart from kernel and initial RAM file system, a virtual machine has one or more associated virtual disks: the actual virtual machine image, a temporary disk image used as writable disk layer – unless a RAM disk is used as writable layer – and possibly a separate temporary disk image used as swap partition. All of these virtual disks may be partitions on physical devices, LVM logical volumes, or images files – the choice is left to the XGE configuration.

The virtual machine image is never altered by the virtual machine because it is only used as a read-only layer in a composite image. Consequently, all instances on one execution host using the same virtual machine image can share a single copy (see Section 4.3). When needed for the first time, the virtual machine image is downloaded once, e.g., via BitTorrent, and cached in a local image pool. All writable virtual disks are attached to DRBD devices so that all I/O is intercepted by the DRBD driver. Each virtual machine has a folder that contains (symbolic links to) all its virtual disks. Due to the symbolic links, the virtual disks' names can easily be kept consistent within the cluster, which is important for live migration, without the need for all involved nodes to use the same actual device names, which would impose a large administrative overhead. On virtual machine start, the writable virtual disks are empty.

Each execution host in the XGE cluster runs an *image daemon* (imaged). The image daemon is responsible for downloading and caching the virtual machine images that are used as read-only layers and for managing the DRBD devices.

7.4.3 Migration Process

To initiate the live migration of a virtual machine from a source to a destination host, the XGE running on the head node performs the following steps. It contacts the image daemons on both hosts and instructs them to prepare the migration. The source hosts responds with the port numbers used by the DRBD devices of the corresponding virtual disks. The destination host allocates DRBD devices, attaches LVM logical volumes or disk images, chooses free ports to use for the synchronization, and sends them back to the head node as well.

The head node then communicates the configuration information to both hosts so that they can update their DRBD configuration accordingly and connect the corresponding DRBD endpoints. When the endpoints are connected, the synchronization starts in primary/secondary mode. Finally, once the synchronization is finished, the DRBD devices on the destination host are put into primary mode.

At this point the regular Xen live migration process is started. When the live migration is completed successfully, the XGE instructs the source host to disconnect the DRBD devices corresponding to the migrated virtual machine. This ensures that the DRBD devices on the destination host run in standalone mode and thus with nearly native I/O speed. Additionally, the source host frees all remaining resources of the migrated virtual machine, i.e., devices, ports, and disk images, after being notified about the successful migration by the XGE.

In case of an error, the virtual machine continues to run on the source host. The XGE instructs the source host to put the DRBD device back into standalone mode for optimal performance. Furthermore, the destination host is instructed to free all resources that correspond to the failed migration attempt, i.e., devices, ports, and disk images. Afterwards, a new attempt to migrate the virtual machine can be made.

Although the Xen hypervisor provides DRBD specific hooks that manage the mode transitions (primary/secondary to primary/primary and back) of DRBD based virtual disks automatically during a live migration, these hooks are not used in the implementation. There are two reasons for this decision. First, the implementation should be independent of the hypervisor and should thus not rely on Xen specific features. Second, the association between the source and the target of a migration is dynamically established for the live migration. Consequently, several configuration tasks have to be executed before actually being able to set the devices into primary/primary mode, which require a much finer grained control over the devices. The mode transitions are simply integrated into the implementation as a final step before migration.

7.5 Experimental Results

In this section, the impact of disk synchronization on the total migration time and on the performance of the migrated virtual machine is evaluated in three different measurements: the synchronization of an idle disk image, a live migration of a virtual machine using local storage while it compiles a Linux kernel, and the evaluation of the overhead induced by the DRBD driver in standalone mode.

All measurements have been conducted on a set of cluster nodes on MaRC. Each of the nodes in the cluster contained two Dual-Core AMD Opteron 2216HE CPUs running at 2.4 GHz, 16 GiB RAM, and a 250 GB SATA hard disk. The nodes were interconnected using a 1 Gbit switched Ethernet network. At the time of the measurements, Debian GNU/Linux 4 (Etch) was used as operating system and Xen 3.0.2 was used as hypervisor.

7.5.1 Idle Synchronization

The first measurement is the synchronization time of an idle disk image for several disk image sizes using the DRBD driver. The synchronization times are compared for disk images that are filled with random content from `/dev/urandom` and empty sparse disk images that are observed as zero filled images by the synchronization driver. The results of this measurement are shown in Figure 7.4. It is evident that the synchronization time grows linearly with the disk size. The results also show that the synchronization time can be reduced by up to 15.6 % when working with sparse images where only a small fraction of the size is actually in use.

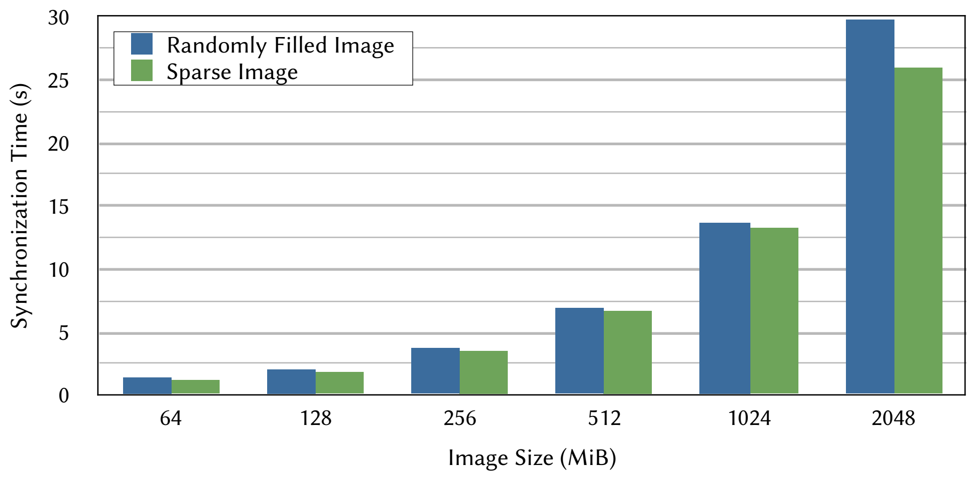


Figure 7.4 Synchronization of Idle Disk Images. This chart shows the synchronization times for disk images in various sizes. For each image size, both a disk image filled with random content and a sparse disk image that appears as zero filled image are compared.

7.5.2 Live Synchronization

The main measurement was the compilation time of a recent Linux kernel with the default configuration. The test was chosen because it represents a balanced workload stressing the virtual memory system, doing moderate disk I/O as well as being relatively CPU intensive. The kernel compilation was done inside a virtual machine that is live migrated to another host at the same time. In Table 7.1 the six slightly different setups that were used for this measurement are listed.

Table 7.1 Measurement Setups. This table lists the six setups used for the kernel compilation measurements.

	RAM (GiB)		
Setup	VM	Host	Writable Layer
1	0.25	1	2 GiB disk image attached to the DRBD driver
2	0.25	13	2 GiB disk image attached to the DRBD driver
3	3	1	2 GiB disk image attached to the DRBD driver
4	3	13	2 GiB disk image attached to the DRBD driver
5	3	1	RAM disk
6	3	13	RAM disk

The different RAM sizes in both the virtual machine and the execution host were chosen to evaluate the relationship between RAM size, performance and synchronization time. In setups 1 to 4, all files generated during the compilation are written to a disk image that is synchronized via the DRBD driver, whereas in setups 5 and 6 the generated files are written to a RAM disk. The metrics used in the measurement are the kernel compilation time, the time required to synchronize the disk images and the time required for the Xen migration. All tests were repeated 50 times to get a robust mean. To get a reference value, the measurement of the kernel compilation time was also done without migrating the virtual machine.

For the measurements including the live migration of the virtual machine, different transfer rates of 5, 10, 20, 40 and 80 MiB/s for the background synchronization were used. Transfer rates of more than 80 MiB/s did not result in a further synchronization speed improvement due to saturation of hardware resources. The hybrid configuration that used a large RAM disk as the writable layer and a large swap partition on a disk image was not tested separately, because from the point of view of the synchronization mechanism there is no difference between a disk image containing a regular file system and a disk image containing a swap partition.

The results for the setups 1 to 4 are shown in Table 7.2 and Figure 7.5, whereas Table 7.3 shows the results for setups 5 and 6.

The total performance degradation in the setups involving disk synchronization compared to the reference values ranges from 0.7 % (in setup 4) to 9.0 % (in setup 1 with a background synchronization transfer rate of 80 Mbit/s). These values refer to a single migration process per kernel compilation. If the values are extrapolated to one migration per hour, the performance degradation ranges from 0.2 % to 1.9 %, which are acceptable values.

The Xen live migration process itself has little impact on the performance of the migrating virtual machine, as indicated when comparing the test results in setups 5 and 6 to their reference values (see Table 7.3). Thus, the disk synchronization very likely causes most of the observed overhead in the other setups.

The migration time does not increase when a RAM disk is used as a writable layer. Consequently, the total migration time is reduced by up to 75 % using a medium background synchronization transfer rate of 40 Mbit/s and by up to 86 % using a lower

Table 7.2 Live Migration Results for Setups 1 to 4. This table shows the compilation time of the Linux kernel as well as the time required to synchronize the disk images and actually migrate the virtual machine using the functionality provided by the hypervisor for the setups 1 to 4 using different background synchronization transfer rates.

			Time required for Task (s) at Synchronization Speed (Mbit/s)				
Setup	Task	Reference	5	10	20	40	80
1	Synchronization	–	400.2	203.0	119.7	104.7	108.5
	Migration	–	8.2	8.7	8.2	8.0	7.5
	Compilation	724.4	741.5	746.5	766.0	795.5	789.5
2	Synchronization	–	398.2	201.7	103.0	52.7	27.0
	Migration	–	7.0	7.2	6.7	6.5	5.7
	Compilation	725.0	736.0	736.7	735.0	735.0	735.2
3	Synchronization	–	397.5	202.0	119.3	102.4	100.3
	Migration	–	34.3	34.4	34.1	34.0	34.3
	Compilation	706.4	714.4	714.2	712.9	717.8	716.8
4	Synchronization	–	401.7	202.6	102.7	52.7	27.3
	Migration	–	32.7	33.0	32.7	32.8	32.7
	Compilation	751.0	756.5	756.4	757.1	757.2	757.5

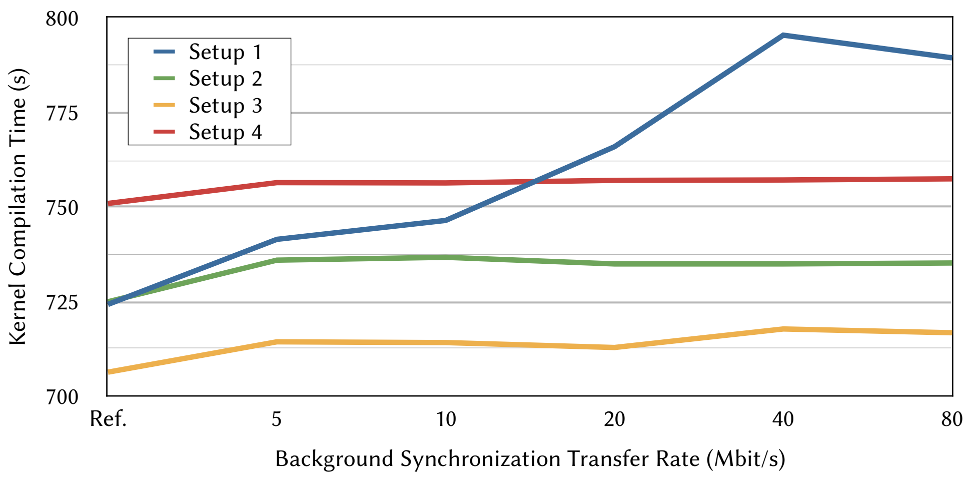


Figure 7.5 Impact of the Live Migration on the Compilation Time. This chart shows the impact of the live migration on the compilation times in setups 1 to 4 for different background synchronization transfer rates compared to the reference.

rate of 10 Mbit/s. Thus, the memory-only configuration is suitable for workloads that do not produce large amounts of data that are written to the virtual disk.

The amount of RAM allocated to the execution host dictates an upper bound to the transfer rate for background synchronization. In the tests with only 1 GB of RAM (see Table 7.2, setups 1 and 3), the synchronization time does not decrease significantly between 40 and 80 Mbit/s, which means that the network bandwidth is higher than the rate at which the incoming data can be processed at the destination host. In

Table 7.3 Live Migration Results for Setups 5 and 6. This table shows the compilation time of the Linux kernel as well as the time required to migrate the virtual machine using the functionality provided by the hypervisor for the setups 5 and 6. Because these two setups are instances of the memory-only configuration, no disk synchronization step is necessary.

Setup	Task	Reference	Migration
5	Migration	–	32.4
	Compilation	722.0	721.0
6	Migration	–	32.9
	Compilation	762.9	765.1

general, the synchronization time is inversely proportional to the transfer rate in a linear fashion.

Increasing the transfer rate for background synchronization has an observable impact only in setup 1, where both the execution host and the virtual machine have a small amount of RAM. This setup represents the most write intensive workload in the sense that fewer writes can be cached in main memory by the kernel (both in the execution host and the virtual machine). In all other cases, the overhead compared to the non-migrating reference values is mostly caused by the synchronous disk writes. Thus, especially for write intensive workloads, choosing the transfer rate will always be a trade-off between the total migration time and the performance impact on the virtual machine.

Strangely enough, in all tests with 3 GB of RAM in the virtual machine, the kernel compilation time increases considerably when allocating more memory to the execution host, e.g., from 714.4 s to 756.5 s in setups 3 and 4, respectively. At the time of writing, no reasonable explanation for this phenomenon could be found, but since these observations do not affect the area under test (disk synchronization), they are simply stated as observed. This topic has to be investigated in more detail in the future.

7.5.3 Runtime Impact of DRBD in Standalone Mode

In normal operation, all disk I/O of the virtual machine goes through the DRBD driver running in standalone mode. The bonnie++ [29] benchmark was used to measure the overhead of the driver. The bonnie++ test was performed both using a disk image file mounted through a *loopback* device and a LVM logical volume. Table 7.4 shows the results of this measurement[2].

The highest performance impact can be observed in the write throughput on the disk image file (decreased by 6.7 %) followed by the read throughput on the LVM logical volume (decreased by 2.4 %). The other two values only differ by around 1 % in both directions. The bonnie++ benchmark is designed for disk performance tests and represents an unusual workload stressing the I/O facilities to a maximum. Average

[2] A description of the bonnie++ tests included in the results can be found in Table 4.10.

Table 7.4 DRDB Overhead in Standalone Mode. This table shows the results of two measurements to assess the overhead induced by DRBD running in standalone mode. The results of the *Character Write* and *Character Read* tests of bonnie++ are reproduced for an image file and a LVM logical volume both for direct access and access through the DRBD driver.

	Character	
Setup	Write (KiB/s)	Read (KiB/s)
Image File	44,003	59,081
Image File (DRBD Standalone)	41,070	59,717
LVM Logical Volume	43,781	57,027
LVM Logical Volume (DRBD Standalone)	43,239	55,634

applications running in a virtual machine usually have a more moderate disk I/O throughput and thus the observable overhead due to the DRBD driver is expected to be much lower.

Especially in comparison with shared storage solutions that are required for live migration if no external solution for storage migration is used, the results indicate that the proposed concept is suitable for Virtualized Grid and Cloud Computing environments like the XGE. For example, the write throughput on a shared NFS file system is decreased by approximately 48 % in synchronous mode and 9 % in asynchronous mode according to [148]. Additionally, the performance of shared storage is expected to decrease with the number of clients using it simultaneously and with higher network utilization. The DRBD approach presented in this chapter allows the virtual machines to do their I/O locally and thus only introduces performance penalties during a live migration process.

7.6 Summary

In this chapter, a concept for synchronizing virtual disks between execution hosts' local storage has been presented that enables the live migration of virtual machines that use locally stored virtual disks. The virtual machine image composition technology presented in Chapter 4 is the foundation of the live migration process. It ensures that the virtual machine image is never modified and can thus be transferred to the destination host at any point in time before the migration. Furthermore, the virtual machine image might already be in the image cache of the destination host and thus does not need to be transferred at all. Only the writable layer that is used as part of the composite image has to be taken care of by the migration process.

Different configurations for composite images are used to implement an efficient live migration process. The first configuration uses a RAM disk as writable layer, such that no disk image needs to be transferred during a live migration. The second configuration uses a temporary disk image of moderate size as writable layer. In this case, the disk image needs to be synchronized during a live migration process. A novel mechanism based on distributed replicated block devices (DRBD) that allows for synchronous writes on the source and the destination host has been presented for this purpose. The evaluation has shown satisfactory performance of this concept.

"Science never solves a problem without creating ten more."
George Bernard Shaw (1856–1950)

8 Conclusion

8.1 Summary

In this thesis, the idea of lifecycle management has been applied to virtual machines used in Virtualized Grid and Cloud Computing environments. Contrary to other use cases of virtual machines, their on-demand execution has been identified as a key attribute of these environments. Based on this attribute, a lifecycle for virtual machines in these environments has been developed. An analysis of this lifecycle has led to the identification of four phases that are insufficiently supported by existing tools and techniques with regard to the requirements of Virtualized Grid and Cloud Computing environments.

A novel approach for disk image provisioning designed for Linux-based virtual machines was presented that is based on the idea of composition. It can be used to decompose virtual machine images into a base layer that contains commonly used parts of the operating system and can be shared between multiple virtual machines as well as a virtual machine specific layer that contains the software unique to it.

A novel repository for Linux-based virtual machine images was presented that is based on the idea of separating the data of the image file, i.e., the contents of the files stored in the image, from its metadata, i.e., the directory hierarchy, file names, and file attributes. This separation is done during an import process and facilitates both an efficient storage of virtual machines and novel ways to deal with virtual machines, e.g., version control and operations working on the metadata level.

Four more proposals have been made to improve both the security of individual virtual machines and the security of the Virtualized Grid and Cloud Computing environments these virtual machines are used in. The first proposal is a novel approach to detect available software updates for virtual machines irrespective of their current state, i.e., whether they are running or dormant. The second proposal is a novel approach to efficiently install software updates in a large number of virtual machines. The third proposal is a method to detect additional types of vulnerabilities, e.g., misconfigurations or insecure services, in virtual machines as part of their continuous maintenance. The fourth proposal is a novel concept for security monitoring on every layer of virtual environments.

Finally, a novel approach for synchronizing the virtual disks of a virtual machine while it is migrated to another host is presented. This approach facilitates migrating virtual machines in the absence of a shared storage system accessible by both the source and the destination host.

Overall, this thesis has presented several solutions to improve the support for the on-demand execution model that is prevalent in Virtualized Grid and Cloud Computing environments in the areas of virtual machine storage, maintenance, deployment and execution.

8.2 Future Work

There are several areas of future research in the area of lifecycle management in Virtualized Grid and Cloud Computing environments, including extensions of the concepts and tools presented in this thesis. These areas are discussed briefly in the following.

Virtual Machine Image Composition

Although the Image Compositor works on even the newest Debian GNU/Linux release, it does lack support for recent developments in Linux. The union mount implementation supported by the Image Compositor (aufs) has never made it into the Linux kernel and is thus depending on regular porting to newer kernels by developers of the Linux distributions. On the other hand, OverlayFS has been integrated in the Linux kernel in December 2014 and thus adding support for it in the Image Compositor is beneficial.

Furthermore, systemd proceeds to replace the classical System V style init system as the init system in most Linux distributions. Because error free boot and shutdown processes using composite disk images rely on a few modifications to these processes, e.g., modifications to init scripts, the impact of the switch to systemd is not yet foreseeable and is thus left as an area of future research. The benefit of a working systemd version would be compatibility with other distributions than Debian GNU/Linux.

Finally, the current implementation of the Image Compositor requires that the mount point of the writable layer is moved into the composite disk image to enable a clean unmount of the root file system. Direct access to the layer is prevented only by the regular access control mechanisms. It may be possible to either improve the protection of the writable layer to prevent even the root user from modifying its contents or an alternative way to cleanly unmount the root file system that does not rely on the writable layer being accessible from within the composite disk image.

Virtual Machine Image Storage

There two areas where the Marvin Image Store can be improved. The first area is the storage back end. The current implementation stores the content of each file as an individual file on a regular file system. Because the number of files increases rapidly when virtual machines are imported[1], the performance of the Image Store depends on the capability of the file system to handle large numbers of files in a small number of directories.

A custom storage system might be a useful extension of the Image Store: a single (or a few) append only data file(s) and a corresponding custom tree structure used as an index into the data file(s) can solve multiple problems related to the file systems at once. Such a storage system would not waste disk space for incompletely filled file system blocks or superfluous file system metadata.

The second area is the Direct Mount feature of the Image Store. The current implementation is restricted to read-only access, whereas the usability of the feature would be greatly improved by write access. A few prototypical extensions have proven that write access is possible and can be used for simple changes to virtual machines. Another possible extension is a kernel module that implements the direct mount feature. This would not only improve the access efficiency, but rather be the first step towards booting a virtual machine directly out of the Image Store without exporting its disk image first.

Virtual Machine Security

The Update Checker currently bases its assessment of an update's priority solely on the repository it is available in. A better way to assess the priority would be to automatically parse the security advisories published by most Linux distributions and derive the priority of updates based on the information in the security advisories. Automated process of security advisories can also be combined with the Image Store to quickly identify affected virtual machines, even if they are not registered with the Update Checker.

There are a few areas for further research related to the centralized update process. The current implementation of the tools that support the centralized update process are only working on the level of package databases. Thus, there might be cases where no masking incompatibilities are detected, but individual files of the update are hidden anyway. The check needs to be extended to the file level to ensure that all updated files are visible.

[1] Table 5.11 lists the number of files for the sets of virtual machines used in the evaluation of the Image Store. However, the 6,045,191 files in total (all six sets combined) are reduced to 690,804 unique files in the back end.

Additionally, updates that change configuration files are problematic if the configuration file has also been changed in the user layer. Automatic merging of those changes into the configuration files of the user layer would greatly improve the applicability of the process. This would require detecting relevant scripts in the updated package and executing those scripts in the context of the composite disk image.

Finally, if users take care of updating installed software themselves, an update might be installed both in the user and the base layer. This is not an actual problem as long as the version in the user layer is up-to-date, but it prevents later updates in the base layer from being visible and wastes disk space. An automatic scan for and removal of such duplicates could solve both problems.

Automatic vulnerability scans suffer from the long scanning times that make their frequent use unlikely for large numbers of virtual machines. Using the information about virtual machines that is available either in the databases of the Update Checker or in the virtual machines' manifests stored in the Image Store, it should be possible to automatically decide for which virtual machines additional vulnerability scans are sensible and which virtual machines can be omitted from those scans.

The proposed concept for a security monitoring system is just the foundation for effective security monitoring. There are three areas where additional research and implementation work is required to make the system usable in practice: the EPAs, the Sensor Framework, and the Action Framework. In the first area, EPAs need to be created for known, common attacks. More importantly, support for detecting anomalies related to unknown attacks and automatically deriving EPAs from historic data are required.

The Sensor Framework is crucial to the detection of anomalies. Novel sensors have to be developed that generate meaningful events and thus make the detection of unknown attacks possible. Finally, actions need to be developed as response to detected attacks that are both effective and appropriate. This is of particular importance for unknown attacks, where generic actions might be able to prevent greater damage, although they are not tailored to a specific attack.

Virtual Machine Migration

The current implementation of the migration technique is tailored to the Xen hypervisor. The underlying technologies, i.e., image composition and storage synchronization, are hypervisor-independent, so porting the migration technique to other hypervisors, e.g., KVM) is beneficial to increase its usability.

A

Virtual Machine Image Storage – Detailed Evaluation Results

Table A.1 Virtual Machines of Set *B* – Details. This table describes the six groups of appliances in more detail. It lists the minimum, maximum, and average values for the number of files contained in the group's virtual machine images, the content size of the group's images, the actual size – disk usage – of the group's images both for the original (if available) and optimized versions, and the apparent size.

				Size (MiB)											
	Number of Files			Content			Image File			Optimized Image File			File System		
ID	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
B1	17,432	47,858	26,998.3	474.7	1,635.4	754.3	1,190.1	2,344.6	1,450.6	668.2	1,828.8	947.7	4,096.0	4,096.0	4,096.0
B2	17,548	51,466	28,744.7	496.1	1,931.1	838.6	1,344.7	2,573.0	1,700.2	689.4	2,124.5	1,031.9	4,096.0	4,096.0	4,096.0
B3	13,662	57,629	29,954.1	319.3	1,747.3	827.8	–	–	–	408.7	2,222.3	986.6	17,408.0	17,408.0	17,408.0
B4	20,874	64,227	34,071.2	540.7	1,969.0	1,036.2	–	–	–	587.3	2,434.3	1,193.8	16,384.0	16,514.0	16,487.7
B5	19,189	47,347	31,765.3	625.2	1,936.6	1,143.7	–	–	–	710.8	2,370.5	1,287.6	5,120.0	17,408.0	8,806.4
B6	23,013	49,844	31,284.2	476.3	945.6	640.7	–	–	–	527.0	1,010.8	702.0	20,480.0	20,480.0	20,480.0

Table A.2 Detailed Results of the Compression Algorithm Comparison. Comparison of the apparent size and disk usage of the Data Store after importing the virtual machine image *A31* and its import time for the compressing back ends. All values are relative to the baseline value generated by importing the same image using the default back end. *Size* is the sum of apparent sizes of all files in the Data Store, *Usage* is the sum of bytes allocated for all files, and *Time* is the import time of the virtual machine image. *4K Thr.* indicates whether the back end has a 4,096 byte threshold or not, i.e., whether it compresses files less than or equal to 4,096 bytes or not.

			Compression Level									
Measure-ment	Algo-rithm	4K Thr.	0	1	2	3	4	5	6	7	8	9
Size	zlib	No	100.18 %	27.98 %	27.12 %	26.47 %	25.71 %	24.94 %	24.47 %	24.35 %	24.20 %	24.14 %
		Yes	100.10 %	28.38 %	27.52 %	26.88 %	26.13 %	25.36 %	24.89 %	24.77 %	24.61 %	24.56 %
	bzip	No	–	22.87 %	22.42 %	22.24 %	22.14 %	22.07 %	22.03 %	21.99 %	21.96 %	21.94 %
		Yes	–	23.23 %	22.78 %	22.60 %	22.50 %	22.43 %	22.39 %	22.35 %	22.32 %	22.30 %
	lzma	No	20.40 %	19.66 %	19.35 %	19.21 %	18.73 %	18.01 %	17.77 %	17.73 %	17.71 %	17.70 %
		Yes	20.84 %	20.11 %	19.79 %	19.65 %	19.18 %	18.47 %	18.22 %	18.19 %	18.16 %	18.16 %
Usage	zlib	No	100.11 %	30.86 %	30.04 %	29.41 %	28.70 %	27.96 %	27.52 %	27.40 %	27.25 %	27.20 %
		Yes	100.08 %	30.85 %	30.04 %	29.41 %	28.69 %	27.95 %	27.51 %	27.39 %	27.24 %	27.19 %
	bzip	No	–	25.94 %	25.50 %	25.32 %	25.23 %	25.16 %	25.12 %	25.09 %	25.06 %	25.04 %
		Yes	–	25.90 %	25.47 %	25.29 %	25.19 %	25.13 %	25.08 %	25.05 %	25.02 %	25.00 %
	lzma	No	23.56 %	22.85 %	22.56 %	22.42 %	21.97 %	21.27 %	21.03 %	21.00 %	20.97 %	20.97 %
		Yes	23.56 %	22.85 %	22.55 %	22.42 %	21.96 %	21.27 %	21.02 %	20.99 %	20.97 %	20.96 %
Time	zlib	No	125.77 %	125.59 %	126.88 %	135.20 %	141.69 %	154.46 %	178.78 %	204.17 %	370.54 %	732.59 %
		Yes	124.17 %	122.35 %	123.25 %	128.47 %	136.63 %	148.78 %	173.56 %	197.39 %	362.44 %	727.44 %
	bzip	No	–	401.19 %	419.47 %	444.86 %	455.87 %	467.98 %	484.19 %	494.08 %	510.62 %	520.54 %
		Yes	–	392.78 %	407.40 %	431.73 %	443.91 %	457.87 %	472.29 %	482.36 %	500.49 %	510.12 %
	lzma	No	256.50 %	302.04 %	361.45 %	451.54 %	1,469.80 %	2,435.99 %	2,692.87 %	3,736.44 %	5,757.83 %	5,793.02 %
		Yes	243.71 %	282.66 %	335.19 %	407.05 %	1,198.06 %	1,907.27 %	2,174.09 %	2,691.33 %	3,709.24 %	3,711.40 %

Table A.3 Analysis of the Compressed Files in the Data Store. This table lists the number of files of virtual machine *A31* whose apparent size or disk usage (Metric) was reduced (-), stayed the same (=), or was increased (+) when they are stored in the Data Store using the zlib, bzip, and lzma back ends with the compression levels 1, 3, and 5. The files are divided in 6 categories based on their original size.

		File Size in KiB																				
		< 4			4 - 16			16 - 64			64 - 256			256 - 1,024			> 1,024			Aggregate		
Metric	Back End	-	=	+	-	=	+	-	=	+	-	=	+	-	=	+	-	=	+	-	=	+
Size	zlib1	8,523	15	4,506	6,220	0	644	4,115	0	145	1,129	0	38	378	0	5	193	0	0	20,558	15	5,338
Usage	zlib1	0	13,011	33	6,162	697	6	4,018	135	2	1,267	0	0	387	0	0	193	0	0	12,027	13,843	41
Size	zlib3	8,528	10	4,506	6,220	0	644	4,115	0	145	1,129	0	38	378	0	5	193	0	0	20,563	10	5,338
Usage	zlib3	0	13,011	33	6,166	693	6	4,018	135	2	1,267	0	0	387	0	0	193	0	0	12,031	13,839	41
Size	zlib5	8,548	12	4,484	6,220	0	644	4,115	0	145	1,129	0	38	378	0	5	193	0	0	20,583	12	5,316
Usage	zlib5	0	13,011	33	6,176	683	6	4,018	135	2	1,267	0	0	387	0	0	193	0	0	12,041	13,829	41
Size	bzip1	9,010	5	4,029	6,207	0	657	4,112	0	148	1,128	0	39	378	0	5	192	0	1	21,027	5	4,879
Usage	bzip1	0	12,894	150	6,165	671	29	4,018	123	14	1,254	13	0	383	4	0	192	0	1	12,012	13,705	194
Size	bzip3	9,010	5	4,029	6,207	0	657	4,112	0	148	1,128	0	39	378	0	5	192	0	1	21,027	5	4,879
Usage	bzip3	0	12,894	150	6,165	671	29	4,018	123	14	1,258	9	0	384	3	0	192	0	1	12,017	13,700	194
Size	bzip5	9,010	5	4,029	6,207	0	657	4,112	0	148	1,128	0	39	378	0	5	192	0	1	21,027	5	4,879
Usage	bzip5	0	12,894	150	6,165	671	29	4,018	123	14	1,258	9	0	384	3	0	192	0	1	12,017	13,700	194
Size	lzma1	8,819	8	4,217	6,222	0	642	4,113	0	147	1,129	0	38	378	0	5	193	0	0	20,854	8	5,049
Usage	lzma1	0	13,025	19	6,195	666	4	4,018	135	2	1,267	0	0	387	0	0	193	0	0	12,060	13,826	25
Size	lzma3	8,819	8	4,217	6,222	0	642	4,113	0	147	1,129	0	38	378	0	5	193	0	0	20,854	8	5,049
Usage	lzma3	0	13,025	19	6,196	665	4	4,018	135	2	1,267	0	0	387	0	0	193	0	0	12,061	13,825	25
Size	lzma5	8,849	9	4,186	6,222	0	642	4,113	0	147	1,129	0	38	378	0	5	193	0	0	20,884	9	5,018
Usage	lzma5	0	13,025	19	6,198	663	4	4,018	135	2	1,267	0	0	387	0	0	193	0	0	12,063	13,823	25

Table A.4 Analysis of the Compressed Files in the Data Store (Threshold). This table lists the number of files of virtual machine *A31* whose apparent size or disk usage (Metric) was reduced (-), stayed the same (=), or was increased (+) when they are stored in the Data Store using the zlib, bzip, and lzma back ends with the compression levels 1, 3, and 5. The files are divided in 6 categories based on their original size.

		File Size in KiB																				
		< 4			4 - 16			16 - 64			64 - 256			256 - 1,024			> 1,024			Aggregate		
Metric	Back End	-	=	+	-	=	+	-	=	+	-	=	+	-	=	+	-	=	+	-	=	+
Size	zlib1	0	13,044	0	6,220	0	644	4,115	0	145	1,129	0	38	378	0	5	193	0	0	12,035	13,044	832
Usage	zlib1	0	13,044	0	6,162	697	6	4,018	135	2	1,267	0	0	387	0	0	193	0	0	12,027	13,876	8
Size	zlib3	0	13,044	0	6,220	0	644	4,115	0	145	1,129	0	38	378	0	5	193	0	0	12,035	13,044	832
Usage	zlib3	0	13,044	0	6,166	693	6	4,018	135	2	1,267	0	0	387	0	0	193	0	0	12,031	13,872	8
Size	zlib5	0	13,044	0	6,220	0	644	4,115	0	145	1,129	0	38	378	0	5	193	0	0	12,035	13,044	832
Usage	zlib5	0	13,044	0	6,176	683	6	4,018	135	2	1,267	0	0	387	0	0	193	0	0	12,041	13,862	8
Size	bzip1	0	13,044	0	6,207	0	657	4,112	0	148	1,128	0	39	378	0	5	192	0	1	12,017	13,044	850
Usage	bzip1	0	13,044	0	6,165	671	29	4,018	123	14	1,254	13	0	383	4	0	192	0	1	12,012	13,855	44
Size	bzip3	0	13,044	0	6,207	0	657	4,112	0	148	1,128	0	39	378	0	5	192	0	1	12,017	13,044	850
Usage	bzip3	0	13,044	0	6,165	671	29	4,018	123	14	1,258	9	0	384	3	0	192	0	1	12,017	13,850	44
Size	bzip5	0	13,044	0	6,207	0	657	4,112	0	148	1,128	0	39	378	0	5	192	0	1	12,017	13,044	850
Usage	bzip5	0	13,044	0	6,165	671	29	4,018	123	14	1,258	9	0	384	3	0	192	0	1	12,017	13,850	44
Size	lzma1	0	13,044	0	6,222	0	642	4,113	0	147	1,129	0	38	378	0	5	193	0	0	12,035	13,044	832
Usage	lzma1	0	13,044	0	6,195	666	4	4,018	135	2	1,267	0	0	387	0	0	193	0	0	12,060	13,845	6
Size	lzma3	0	13,044	0	6,222	0	642	4,113	0	147	1,129	0	38	378	0	5	193	0	0	12,035	13,044	832
Usage	lzma3	0	13,044	0	6,196	665	4	4,018	135	2	1,267	0	0	387	0	0	193	0	0	12,061	13,844	6
Size	lzma5	0	13,044	0	6,222	0	642	4,113	0	147	1,129	0	38	378	0	5	193	0	0	12,035	13,044	832
Usage	lzma5	0	13,044	0	6,198	663	4	4,018	135	2	1,267	0	0	387	0	0	193	0	0	12,063	13,842	6

Table A.5 Disk Usage – Regular Images. This table shows the disk usage of each regular virtual machine in set *A* as plain image (Default), optimized image (Opt.), compressed image (Comp.), optimized and compressed image (O. + C.), as well as stored in the Image Store with the selected back ends.

	Disk Usage (MiB)								
	Image				MIS				
ID	Default	Opt.	Comp.	O. + C.	default	zlib1	zlib3	zlib5	lzma1
A01	1,190.1	668.2	706.4	194.2	455.3	218.0	212.8	206.3	184.8
A02	1,214.5	697.3	728.6	209.8	482.8	236.5	231.2	224.3	201.9
A03	1,222.9	699.1	720.0	203.6	485.6	232.0	226.4	219.5	197.3
A04	1,249.3	733.2	721.2	215.2	512.2	248.3	242.6	235.3	211.6
A05	1,278.6	748.5	752.1	224.1	532.9	254.6	248.5	240.9	213.9
A06	1,301.1	774.0	755.7	233.7	556.3	274.5	268.4	260.7	236.8
A07	1,290.7	777.0	742.1	235.2	560.8	268.2	261.6	253.5	225.4
A08	1,298.0	783.2	747.2	230.1	566.1	275.8	269.5	261.6	235.8
A09	1,303.3	790.4	745.4	242.4	572.8	282.5	276.1	268.0	241.8
A10	1,313.2	806.4	752.4	241.5	589.6	274.0	267.5	259.3	229.3
A11	1,302.7	815.3	723.7	235.3	588.0	275.4	269.0	261.0	233.6
A12	1,325.3	819.8	748.0	249.1	598.6	294.3	287.6	279.3	251.5
A13	1,356.0	840.6	757.8	254.9	618.8	302.2	295.2	286.5	257.7
A14	1,341.9	848.6	740.3	254.2	630.1	293.5	286.5	277.7	246.9
A15	1,355.1	875.9	730.8	252.6	641.5	303.9	296.7	287.8	259.2
A16	1,363.1	886.4	729.6	265.8	663.8	307.9	300.7	291.6	260.8
A17	1,382.7	894.0	747.8	267.1	675.9	312.1	304.0	294.1	262.8
A18	1,423.0	898.9	837.1	300.8	679.5	334.2	326.5	316.5	284.1
A19	1,413.8	900.5	785.2	281.9	666.8	312.1	304.1	294.2	260.1
A20	1,443.3	915.6	802.5	273.9	686.5	331.5	323.6	313.8	283.4
A21	1,445.2	930.5	793.2	289.7	701.1	335.5	326.9	316.6	284.5
A22	1,423.7	936.9	783.3	288.0	708.4	338.3	330.4	320.8	287.2
A23	1,471.5	958.6	808.4	304.9	732.2	350.1	341.3	330.5	296.0
A24	1,505.0	990.5	829.1	323.3	767.8	361.3	352.4	341.1	304.4
A25	1,412.6	991.8	729.0	306.3	752.7	359.6	350.4	339.5	306.7
A26	1,634.7	1,127.1	840.3	343.7	879.6	440.0	430.0	417.0	377.8
A27	1,581.7	1,175.8	751.0	351.6	904.4	410.8	399.5	386.4	345.6
A28	1,715.7	1,208.5	898.4	404.6	968.4	468.6	457.4	443.7	400.3
A29	1,904.5	1,393.4	1,021.1	522.1	1,138.3	590.1	577.2	561.3	505.9
A30	2,160.7	1,662.6	1,047.2	557.8	1,421.1	653.4	636.0	613.0	527.7
A31	2,344.6	1,828.8	945.5	445.1	1,601.5	497.4	474.2	451.0	369.5

Table A.6 Disk Usage – Layered Images. This table shows the disk usage of each layered virtual machine in set *A* as plain image (Default), optimized image (Opt.), compressed image (Comp.), optimized and compressed image (O. + C.), as well as stored in the Image Store with the selected back ends.

	Disk Usage (MiB)								
	Image				MIS				
ID	Default	Opt.	Comp.	O. + C.	default	zlib1	zlib3	zlib5	lzma1
A01	1,208.3	668.2	729.1	194.2	454.6	218.3	213.2	206.6	185.1
A02	161.9	144.1	47.9	31.6	45.3	26.4	26.1	25.5	22.8
A03	152.2	146.0	31.4	25.5	47.9	21.7	21.2	20.5	18.1
A04	202.2	180.5	57.1	37.1	75.4	38.9	38.2	37.2	33.2
A05	231.6	195.3	76.8	45.9	95.1	44.3	43.2	41.9	34.6
A06	279.7	221.1	101.3	55.5	116.9	62.8	61.7	60.3	56.1
A07	271.8	226.0	101.4	58.6	125.0	58.8	57.2	55.3	46.8
A08	264.3	230.0	84.3	51.9	126.5	64.0	62.8	61.1	55.1
A09	285.3	237.5	107.3	64.3	133.6	71.0	69.7	67.8	61.4
A10	316.2	253.4	115.4	63.5	151.5	63.5	62.0	60.1	49.8
A11	315.7	262.6	98.6	57.1	148.9	64.0	62.7	60.9	53.2
A12	328.3	267.4	126.3	71.0	159.1	82.6	80.9	78.9	70.9
A13	360.1	287.6	139.9	76.8	179.1	90.4	88.5	86.0	76.9
A14	359.3	295.7	133.4	76.1	191.2	82.2	80.3	77.8	66.7
A15	390.9	323.0	129.0	74.6	201.7	92.0	89.9	87.4	78.4
A16	415.8	333.6	158.6	87.8	224.4	96.2	94.1	91.3	80.2
A17	414.6	341.2	156.1	89.0	236.3	100.4	97.3	93.7	82.1
A18	468.3	352.8	228.5	123.5	245.1	127.2	124.5	120.9	108.0
A19	433.2	348.1	183.6	103.9	228.1	101.0	98.0	94.4	80.1
A20	439.6	363.1	163.1	95.2	246.5	119.7	116.8	113.2	102.5
A21	468.3	378.1	195.7	110.9	262.9	125.0	121.4	117.4	105.0
A22	498.8	384.6	206.9	110.1	269.6	126.6	123.8	120.4	106.5
A23	528.9	415.2	242.6	135.5	301.3	147.1	143.3	138.9	124.1
A24	575.6	444.3	266.9	145.9	332.6	153.6	149.7	144.8	127.6
A25	511.6	439.4	192.3	128.4	312.7	147.7	143.6	139.0	125.8
A26	717.2	574.6	300.4	165.5	439.7	229.0	224.1	217.3	197.9
A27	765.6	623.1	301.7	173.5	476.2	204.9	198.5	191.5	169.6
A28	859.4	665.0	415.1	235.2	537.1	265.9	259.7	252.3	228.5
A29	1,169.2	847.4	650.0	344.9	704.5	384.3	376.4	366.9	331.1
A30	1,448.5	1,110.5	704.0	379.7	983.1	443.9	431.7	415.0	349.4
A31	1,491.4	1,278.6	472.2	268.0	1,164.3	286.9	268.8	251.8	189.9

Table A.7 Import Time – Regular Images. For each regular virtual machine in set *A*, this table shows the import time into the Image Store with the selected back ends and the time required to copy or compress the corresponding image file or to recursively copy its contents.

	Time (s)							
	Copy			MIS Import				
ID	Image	Content	Compress	default	zlib1	zlib3	zlib5	lzma1
A01	18.8	28.7	72.1	37.1	49.2	53.7	58.4	113.8
A02	21.3	32.1	74.0	44.2	56.5	58.7	63.9	123.7
A03	17.0	29.2	72.9	40.6	53.3	57.0	63.0	122.9
A04	17.9	34.9	73.7	50.0	66.1	66.7	72.5	133.1
A05	20.0	33.1	74.8	44.8	57.9	62.8	69.2	131.4
A06	20.3	43.7	76.2	58.3	70.4	73.9	75.9	144.8
A07	19.0	33.9	77.2	49.3	63.0	66.9	71.6	140.0
A08	16.7	35.5	75.9	52.3	70.6	71.7	79.0	139.9
A09	19.2	37.5	75.0	52.0	68.4	70.6	78.1	144.4
A10	23.1	39.4	76.0	55.4	66.0	69.0	75.4	140.3
A11	17.7	39.9	73.3	54.8	67.9	72.9	78.9	146.2
A12	17.9	40.2	75.4	58.1	70.0	76.7	80.8	150.4
A13	17.2	38.7	76.4	57.7	73.6	78.6	85.7	153.7
A14	21.8	41.5	75.6	56.9	73.0	74.6	81.5	149.0
A15	18.9	43.1	75.4	66.5	81.6	82.8	89.6	154.0
A16	23.4	47.0	75.6	60.7	74.3	78.6	84.4	159.9
A17	17.5	43.6	77.1	64.6	81.6	83.0	89.4	161.6
A18	21.5	40.8	80.9	56.3	76.9	78.7	85.4	172.5
A19	22.4	58.0	80.6	75.5	85.0	90.1	86.9	175.6
A20	24.1	57.6	80.7	82.2	92.5	93.6	99.1	178.6
A21	20.3	52.3	78.1	73.1	86.1	93.7	98.5	185.1
A22	19.0	45.7	78.0	68.5	85.1	87.8	94.6	168.8
A23	25.9	63.9	82.2	80.7	91.1	98.0	97.4	188.4
A24	23.6	47.2	81.0	65.3	80.4	86.4	94.3	188.2
A25	22.7	63.9	74.8	84.3	94.3	102.8	109.4	194.6
A26	21.6	76.6	83.4	105.2	113.6	123.5	126.1	225.1
A27	26.0	81.6	78.9	104.4	113.7	122.9	124.3	225.3
A28	27.2	87.6	88.9	110.0	123.0	129.2	132.6	256.3
A29	26.3	82.2	96.7	107.7	128.9	139.9	145.9	304.1
A30	41.5	116.1	103.4	147.9	164.0	173.8	175.5	345.2
A31	30.8	71.6	96.3	104.2	114.7	123.8	140.8	261.0

Table A.8 Import Time – Layered Images. For each layered virtual machine in set *A*, this table shows the import time into the Image Store with the selected back ends and the time required to copy or compress the corresponding image file or to recursively copy its contents.

	Time (s)							
	Copy			MIS Import				
ID	Image	Content	Compress	default	zlib1	zlib3	zlib5	lzma1
A01	18.4	28.3	73.9	37.1	50.3	53.9	61.3	117.1
A02	5.1	3.0	19.8	3.7	5.0	5.3	5.9	12.5
A03	4.1	3.1	18.4	4.0	5.2	5.4	6.2	11.6
A04	5.1	6.5	20.1	9.1	10.7	11.5	12.4	20.7
A05	5.4	4.6	21.1	5.9	8.1	8.8	10.2	20.8
A06	6.3	9.1	22.3	11.7	14.1	14.8	16.2	28.7
A07	7.1	7.7	22.9	9.0	12.0	13.0	14.6	27.2
A08	5.2	8.0	21.4	11.6	14.2	14.9	16.4	30.6
A09	5.7	8.5	22.7	11.4	14.8	15.8	17.3	33.6
A10	7.0	6.4	23.7	8.0	11.3	12.4	14.3	30.3
A11	5.5	9.1	22.2	11.9	15.0	16.1	17.9	32.9
A12	5.8	9.8	23.4	12.9	16.7	17.8	19.6	36.8
A13	6.4	10.0	24.3	13.9	17.8	18.8	21.1	41.7
A14	8.4	11.3	24.7	13.9	17.7	19.1	21.2	41.1
A15	7.3	14.3	24.4	18.8	22.9	24.1	26.2	53.7
A16	6.4	10.6	25.1	14.1	18.6	19.9	22.6	45.9
A17	8.3	13.0	25.7	16.6	21.4	23.1	26.1	49.1
A18	7.6	10.6	28.8	13.7	19.7	20.9	24.1	65.4
A19	7.6	12.0	27.2	14.4	20.5	22.3	25.8	51.2
A20	9.7	21.9	26.5	27.4	32.8	34.5	37.6	61.2
A21	8.5	18.7	28.0	25.1	31.8	33.7	37.5	66.1
A22	8.2	15.1	28.1	20.3	25.8	27.4	30.8	59.0
A23	8.9	20.2	30.4	24.8	32.4	34.8	40.3	75.0
A24	9.4	13.9	30.9	18.4	26.5	28.6	32.9	82.5
A25	9.3	27.7	27.6	35.2	44.1	46.0	50.1	77.3
A26	11.4	34.1	34.3	47.9	58.5	61.9	67.7	105.0
A27	13.2	33.7	35.4	46.4	56.0	58.3	64.9	103.6
A28	14.1	40.4	42.9	52.0	65.7	69.1	75.3	140.9
A29	14.3	41.4	55.3	54.4	76.8	78.7	88.2	190.4
A30	15.8	47.0	61.4	73.8	97.7	101.1	113.3	212.6
A31	15.6	28.1	45.4	38.3	61.4	68.2	79.6	156.4

Table A.9 Export Time – Regular Images. For each regular virtual machine in set *A*, this table shows the export time out of the Image Store with the selected back ends and the time required to copy or decompress the corresponding image file or to recursively copy its contents.

	Time (s)							
	Copy			MIS Export				
ID	Image	Content	Extract	default	zlib1	zlib3	zlib5	lzma1
A01	18.8	28.7	50.0	34.0	41.3	41.0	41.2	43.1
A02	21.3	32.1	44.8	40.5	49.5	48.0	46.1	47.7
A03	17.0	29.2	44.7	37.3	47.2	47.5	45.7	46.2
A04	17.9	34.9	48.6	47.5	58.9	58.0	53.5	56.6
A05	20.0	33.1	48.8	39.1	49.0	48.8	48.3	50.0
A06	20.3	43.7	42.1	43.1	57.3	56.7	57.4	57.0
A07	19.0	33.9	45.1	40.1	51.7	51.9	50.0	55.7
A08	16.7	35.5	44.7	53.4	63.7	61.9	59.6	58.2
A09	19.2	37.5	50.1	50.3	62.5	60.3	58.1	59.0
A10	23.1	39.4	45.4	49.1	54.3	53.0	52.0	53.3
A11	17.7	39.9	49.6	47.0	58.8	59.8	59.5	58.6
A12	17.9	40.2	44.8	48.6	60.5	62.2	62.0	62.1
A13	17.2	38.7	45.4	48.7	68.2	64.4	63.4	65.8
A14	21.8	41.5	44.5	52.4	63.7	61.0	59.1	59.6
A15	18.9	43.1	44.5	60.1	72.0	69.6	69.5	65.3
A16	23.4	47.0	45.7	47.8	67.5	65.2	63.7	66.7
A17	17.5	43.6	45.8	56.5	69.0	67.9	65.7	65.5
A18	21.5	40.8	44.3	51.0	62.2	59.1	55.9	64.3
A19	22.4	58.0	46.0	45.2	59.8	58.8	61.1	66.8
A20	24.1	57.6	47.1	67.1	80.8	81.1	78.5	79.8
A21	20.3	52.3	44.4	57.4	76.9	81.6	76.9	79.6
A22	19.0	45.7	46.3	60.1	74.6	72.2	72.0	70.8
A23	25.9	63.9	52.3	55.3	76.7	76.1	76.0	77.7
A24	23.6	47.2	53.3	48.3	65.2	67.6	65.4	74.2
A25	22.7	63.9	46.8	62.0	87.4	89.5	86.9	89.7
A26	21.6	76.6	47.9	80.5	105.0	108.0	104.4	111.5
A27	26.0	81.6	52.0	74.9	99.7	103.9	99.4	100.4
A28	27.2	87.6	49.9	78.0	110.9	107.8	110.2	108.0
A29	26.3	82.2	54.1	83.6	119.6	123.0	116.5	128.5
A30	41.5	116.1	56.9	113.5	143.8	143.5	132.7	149.1
A31	30.8	71.6	58.8	78.0	99.6	99.8	93.8	104.4

Table A.10 Export Time – Layered Images. For each layered virtual machine in set *A*, this table shows the export time out of the Image Store with the selected back ends and the time required to copy or decompress the corresponding image file or to recursively copy its contents.

	Time (s)							
	Copy			MIS				
ID	Image	Content	Extract	default	zlib1	zlib3	zlib5	lzma1
A01	18.4	28.3	42.4	34.8	42.8	41.6	42.3	44.7
A02	5.1	3.0	16.9	3.0	4.0	3.9	3.8	4.7
A03	4.1	3.1	17.6	3.5	4.5	4.2	4.2	4.7
A04	5.1	6.5	16.5	9.6	10.1	10.0	10.0	10.2
A05	5.4	4.6	16.5	5.3	6.4	6.3	6.4	6.8
A06	6.3	9.1	18.5	10.8	12.8	12.8	12.6	12.2
A07	7.1	7.7	17.0	6.9	8.4	8.5	8.4	9.1
A08	5.2	8.0	15.9	12.6	13.3	13.0	13.0	14.3
A09	5.7	8.5	17.4	11.4	12.7	12.6	12.6	13.0
A10	7.0	6.4	18.4	7.5	8.2	8.2	8.4	9.6
A11	5.5	9.1	18.2	12.3	14.2	13.6	13.6	13.4
A12	5.8	9.8	18.6	12.9	15.6	14.8	14.8	14.8
A13	6.4	10.0	18.5	14.6	17.1	17.0	16.7	16.3
A14	8.4	11.3	20.0	11.8	12.9	12.9	12.9	15.5
A15	7.3	14.3	17.4	18.8	20.5	20.1	19.9	26.6
A16	6.4	10.6	17.8	13.9	16.5	16.6	16.9	18.1
A17	8.3	13.0	20.5	15.2	17.3	17.3	16.9	17.6
A18	7.6	10.6	18.1	12.2	13.9	13.8	13.6	19.8
A19	7.6	12.0	19.5	11.4	14.6	14.9	14.1	16.1
A20	9.7	21.9	18.7	23.1	26.0	25.6	25.3	25.6
A21	8.5	18.7	18.4	22.1	25.4	24.5	24.5	25.0
A22	8.2	15.1	18.3	19.2	21.0	21.3	21.3	21.6
A23	8.9	20.2	19.6	19.9	23.8	23.7	23.2	23.8
A24	9.4	13.9	20.0	16.3	19.9	19.8	19.6	26.2
A25	9.3	27.7	18.7	25.3	32.4	29.7	29.2	29.9
A26	11.4	34.1	20.3	43.2	50.2	46.3	46.3	48.2
A27	13.2	33.7	21.7	37.3	42.4	41.7	41.5	40.4
A28	14.1	40.4	21.3	41.5	52.0	51.6	50.7	53.6
A29	14.3	41.4	23.7	48.6	62.8	60.4	59.7	71.0
A30	15.8	47.0	26.1	61.7	82.0	77.5	77.0	84.5
A31	15.6	28.1	22.7	38.4	41.1	40.4	39.6	45.4

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

List of Listings

Bibliography

[1] Yoshihisa Abe, Roxana Geambasu, Kaustubh Joshi, H. Andrés Lagar-Cavilla, and Mahadev Satyanarayanan. vTube: Efficient Streaming of Virtual Appliances over Last-mile Networks. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*, pages 16:1–16:16, New York, NY, USA, 2013. ACM.

[2] ACCEPT Project. ACCEPT – Mastering Security Anomalies in Virtualized Computing Environments via Complex Event Processing. `http://www.accept-projekt.de`, 2015.

[3] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, pages 2–13, New York, NY, USA, 2006. ACM.

[4] Stephen Adler. The Slashdot Effect, An Analysisy of Three Internet Publications. *Linux Gazette*, 38, 2009.

[5] The AIDE Project. Advanced Intrusion Detection Environment (AIDE). `http://aide.sourceforge.net/`, 2011.

[6] Samer Al-Kiswany, Dinesh Subhraveti, Prasenjit Sarkar, and Matei Ripeanu. VMFlock: Virtual Machine Co-migration for the Cloud. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC '11)*, pages 159–170, New York, NY, USA, 2011. ACM.

[7] Marco Aldinucci, Massimo Torquati, Marco Vanneschi, and Pierfrancesco Zuccato. The VirtuaLinux Storage Abstraction Layer for Efficient Virtual Clustering. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 619–627. IEEE Computer Society, 2008.

[8] Amazon Web Services, Inc. Amazon Web Services (AWS) - Cloud Computing Services. `http://aws.amazon.com/`, August 2014.

[9] Zach Amsden, Daniel Arai, Daniel Hecht, Anne Holler, and Pratap Subrahmanyam. VMI: An Interface for Paravirtualization. In *Proceedings of the Ottawa Linux Symposium 2006*, volume 2, pages 363–378, 2006.

[10] Valerie Aurora. Unioning File Systems: Architecture, Features, and Design Choices. *LWN*, March 2009. `http://lwn.net/Articles/324291/`.

[11] Valerie Aurora. Unioning file systems: Implementations (Part I and II). *LWN*, April 2009. `http://lwn.net/Articles/325369/` and `http://lwn.net/Articles/327738/`.

[12] Edward C. Bailey, Paul Nasrat, Matthias Saou, and Ville Skyttä. Maximum RPM - Taking the RPM Package Manager to the Limit. `http://www.rpm.org/max-rpm-snapshot/`, 2000.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, October 2003.

[14] Mick Bauer. Paranoid Penguin: An Introduction to Novell AppArmor. *Linux Journal*, 2006:13, August 2006.

[15] Lars Baumgärtner, Pablo Graubner, Matthias Leinweber, Roland Schwarzkopf, Matthias Schmidt, Bernhard Seeger, and Bernd Freisleben. Mastering Security Anomalies in Virtualized Computing Environments via Complex Event Processing. In *Proceedings of the 4th International Conference on Information, Process, and Knowledge Management (eKNOW 2012)*, pages 76–81. XPS, 2012.

[16] Lars Baumgärtner, Christian Strack, Bastian Hoßbach, Marc Seidemann, Bernhard Seeger, and Bernd Freisleben. Complex Event Processing for Reactive Security Monitoring in Virtualized Computer Systems. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 22–33, New York, NY, USA, 2015. ACM.

[17] BitRock, Inc. Bitnami Stacks. `http://bitnami.com/stacks`, July, 2015.

[18] Biz2Grid Project. Biz2Grid: Moving Business to the Grid - An Application for the Automotive Industry. `http://www.im.uni-karlsruhe.de/biz2grid/index.htm`, 2010.

[19] Sören Bleikertz, Matthias Schunter, Christian W. Probst, Dimitrios Pendarakis, and Konrad Eriksson. Security Audits of Multi-tier Virtual Infrastructures in Public Infrastructure Clouds. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security (CCSW '10)*, pages 93–102, New York, NY, USA, 2010. ACM.

[20] BMBF. Bundesministerium für Bildung und Forschung. `http://www.bmbf.de`, 2015.

[21] Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann. Non-intrusive Virtualization Management Using Libvirt. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*, pages 574–579, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[22] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schöberg. Live Wide-area Migration of Virtual Machines Including Local Persistent State. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07)*, pages 169–179, New York, NY, USA, 2007. ACM.

[23] btrfs authors. Btrfs. `https://btrfs.wiki.kernel.org`, 2015.

[24] Canonical, Ltd. Ubuntu Advantage Landscape. `http://www.canonical.com/enterprise-services/ubuntu-advantage/landscape`, Aug 2011.

[25] David Chadwick. Authorisation in Grid Computing. *Information Security Tech. Report*, 10(1):33–40, January 2005.

[26] Jeffrey S. Chase, David E. Irwin, Laura E. Grit, Justin D. Moore, and Sara E. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC '03)*, pages 90–100, Washington, DC, USA, 2003. IEEE Computer Society.

[27] Han Chen, Minkyong Kim, Zhe Zhang, and Hui Lei. Empirical Study of Application Runtime Performance Using On-demand Streaming Virtual Disks in the Cloud. In *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference (MIDDLEWARE '12)*, pages 5:1–5:6, New York, NY, USA, 2012. ACM.

[28] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation (NSDI'05)*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[29] Russell Coker. bonnie++. `http://www.coker.com.au/bonnie++/`, 2001.

[30] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). `http://www.ietf.org/rfc/rfc4627.txt`, Jul 2006.

[31] D-Grid-Initiative. D-Grid. `http://www.d-grid.de`, 2010.

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

[33] The Debian Project. deb-version(5)- Debian package version number format. `http://man7.org/linux/man-pages/man5/deb-version.5.html`, 2013.

[34] The Debian Project. Multiarch HOWTO. `https://wiki.debian.org/Multiarch/HOWTO`, March 2014.

[35] Peter J. Denning. Third Generation Computer Systems. *ACM Computing Surveys*, 3(4):175–216, December 1971.

[36] Distributed Management Task Force (DMTF). Open Virtualization Format. `http://www.dmtf.org/standards/ovf`, 2015.

[37] Tim Dörnemann, Ernst Juhnke, and Bernd Freisleben. On-Demand Resource Provisioning for BPEL Workflows Using Amazon's Elastic Compute Cloud. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '09)*, pages 140–147. IEEE Press, 2009.

[38] Tim Dornemann, Markus Mathes, Roland Schwarzkopf, Ernst Juhnke, and Bernd Freisleben. DAVO: A Domain-Adaptable, Visual BPEL4WS Orchestrator. In *Proceedings of the 23rd International Conference on Advanced Information Networking and Applications (AINA '09)*, pages 121–128, Washington, DC, USA, May 2009. IEEE Computer Society.

[39] Ralph Droms. RFC 2131. Dynamic Host Configuration Protocol. `http://tools.ietf.org/html/rfc2131`, March 1997.

[40] Jeremy Elson and Jon Howell. Handling Flash Crowds from Your Garage. In *Proceedings of the USENIX 2008 Annual Technical Conference (ATC '08)*, pages 171–184, Berkeley, CA, USA, 2008. USENIX Association.

[41] ENISA - European Network and Information Security Agency. Cloud Computing Risk Assessment. `http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-risk-assessment`, November 2009.

[42] EVMS authors. Enterprise Volume Management System. `http://evms.sourceforge.net/`, February 2006.

[43] Niels Fallenbeck, Matthias Schmidt, Roland Schwarzkopf, and Bernd Freisleben. Inter-Site Virtual Machine Image Transfer in Grids and Clouds. In *Proceedings of the 2nd International ICST Conference on Cloud Computing (CloudComp 2010)*, pages 1–19. Springer LNICST, 2010.

[44] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Publishing Company, Incorporated, 2014.

[45] Massimo Ficco. Achieving Security by Intrusion-Tolerance Based on Event Correlation. *Network Protocols and Algorithms*, 2(3):70–84, 2010.

[46] Renato J. Figueiredo, Peter A. Dinda, and José A. B. Fortes. A Case For Grid Computing On Virtual Machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03)*, pages 550–559, Washington, DC, USA, 2003. IEEE Computer Society.

[47] Filesystem Hierarchy Standard Group. Filesystem Hierarchy Standard. `http://www.pathname.com/fhs/`, January, 2004.

[48] FinGrid Project. FinGrid – Entwicklung und Analyse von Grid Aplikationen für die Finanzdienstleistungsindustrie. `http://www.d-grid-gmbh.de/index.php?id=75`, 2012.

[49] National Institute for Standards and Technology. The NIST Definition of Cloud Computing. Recommendations of the National Institute of Standards and Technology Special Publication 800 - 145, April 2012.

[50] National Institute for Standards and Technology. Secure Hash Standard (SHS). Federal Information Processing Standards Publication FIPS Pub 180-4, August 2015.

[51] Ian Foster. What is the Grid? A Three Point Checklist. *GRID Today*, 1(6):32–36, July 2002.

[52] Ian Foster. Globus Toolkit Version 4: Software for Service-oriented Systems. *Journal of Computer Science and Technology*, 21(4):513–520, 2006.

[53] Ian Foster, Tim Freeman, Kate Keahey, Doug Scheftner, Borja Sotomayor, and Xuehai Zhang. Virtual Clusters for Grid Communities. In *Proceedings of the 6h IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06)*, pages 513–520, Washington, DC, USA, 2006. IEEE Computer Society.

[54] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[55] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Virtual Scalable Organizations. *International Journal of High Performance Computing Applications*, 15:220–222, 2001.

[56] Free Software Foundation, Inc. GNU stroke – Dynamically Changing Modification, Access, and Change Time Components. `http://stroke.sourceforge.net/`, January 2012.

[57] Sadayuki Furuhashi. MessagePack – It's like JSON. but fast and small. `http://msgpack.org`, 2013.

[58] FUSE authors. FUSE: Filesystem in Userspace. `http://fuse.sourceforge.net/`, May, 2015.

[59] Tal Garfinkel and Mendel Rosenblum. When Virtual is Harder Than Real: Security Challenges in Virtual Machine Based Computing Environments. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems (HOTOS'05)*, pages 20–20, Berkeley, CA, USA, 2005. USENIX Association.

[60] Robert P. Goldberg. *Architectural Principles for Virtual Computer Systems.* PhD thesis, Division of Engineering and Applied Physics, Harvard University, Cambridge, Massachusetts, 1972.

[61] Dan Gorton. Extending Intrusion Detection with Alert Correlation and Intrusion Tolerance. *Licentiate Thesis, Chalmers University of Technology*, 2003.

[62] Pablo Graubner, Matthias Schmidt, and Bernd Freisleben. Energy-efficient Management of Virtual Machines in Eucalyptus. In *Proceedings of the 4th IEEE International Conference on Cloud Computing (CLOUD 2011)*, pages 243–250. IEEE Press, July 2011.

[63] GRUB authors. GNU GRUB. `http://www.gnu.org/software/grub/`, December 2012.

[64] Kiryong Ha, Padmanabhan Pillai, Wolfgang Richter, Yoshihisa Abe, and Mahadev Satyanarayanan. Just-in-time Provisioning for Cyber Foraging. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '13)*, pages 153–166, New York, NY, USA, 2013. ACM.

[65] Katharina Haselhorst, Matthias Schmidt, Roland Schwarzkopf, Niels Fallenbeck, and Bernd Freisleben. Efficient Storage Synchronization for Live Migration in Cloud Infrastructures. In *Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '11)*, pages 511–518, Washington, DC, USA, 2011. IEEE Computer Society.

[66] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy Live Migration of Virtual Machines. *ACM SIGOPS Operating System Review*, 43(3):14–26, 2009.

[67] HMWK. Hessisches Ministerium für Wissenschaft und Kunst. `http://wissenschaft.hessen.de`, 2015.

[68] Harald Hoyer. dracut. `https://www.kernel.org/pub/linux/utils/boot/dracut/dracut.html`, October 2013.

[69] Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra. *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things.* Morgan Kaufmann. Morgan Kaufmann, October 2011.

[70] IETF Secure Shell Working Group. Secure Shell (secsh). `http://datatracker.ietf.org/wg/secsh/`, March 2007.

[71] IETF Transport Layer Security Working Group. Transport Layer Security (tls). `http://datatracker.ietf.org/wg/tls/`, July 2015.

[72] K. R. Jayaram, Chunyi Peng, Zhe Zhang, Minkyong Kim, Han Chen, and Hui Lei. An Empirical Analysis of Similarity in Virtual Machine Images. In *Proceedings of the Middleware 2011 Industry Track Workshop (Middleware '11)*, pages 6:1–6:6, New York, NY, USA, 2011. ACM.

[73] Deepak Jeswani, Manish Gupta, Pradipta De, Aarpit Malani, and Umesh Bellur. Minimizing Latency in Serving Requests through Differential Template Caching in a Cloud. In *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD 2012)*, pages 269–276. IEEE Press, June 2012.

[74] Keren Jin and Ethan L. Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference (SYSTOR '09)*, pages 7:1–7:12, New York, NY, USA, 2009. ACM.

[75] M. Tim Jones. Inside the Linux boot process. IBM Developer Works Article. `http://www.ibm.com/developerworks/library/l-linuxboot/index.html`, May 2006.

[76] Ernst Juhnke, Tim Dörnemann, Roland Schwarzkopf, and Bernd Freisleben. Security, Fault Tolerance and Modeling of Grid Workflows in BPEL4Grid. In *Proceedings of Software Engineering 2010, Grid Workflow Workshop (GWW 2010), Lecture Notes in Informatics (LNI), Vol. P-160*, pages 193–200. Gesellschaft für Informatik (GI), 2010.

[77] Katarzyna Keahey and Tim Freeman. Contextualization: Providing One-Click Virtual Clusters. In *Proceedings of the 4th IEEE International Conference on*

eScience (ESCIENCE '08), pages 301–308, Washington, DC, USA, Dec 2008. IEEE Computer Society.

[78] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. A Formal Investigation of Diff3. In *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*, pages 485–496, Berlin, Heidelberg, 2007. Springer-Verlag.

[79] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Ottawa Linux Symposium 2007*, volume 1, pages 225–230, 2007.

[80] Evangelos Kotsovinos, Tim Moreton, Ian Pratt, Russ Ross, Keir Fraser, Steven Hand, and Tim Harris. Global-scale Service Deployment in the XenoServer Platform. In *Proceedings of the 1st Workshop on Real, Large Distributed Systems (WORLDS '04)*, Berkeley, CA, USA, 2004. USENIX Association.

[81] Michael Kozuch and Mahadev Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications*, pages 40–46, Washington, DC, USA, 2002. IEEE Computer Society.

[82] Rob Landley. ramfs, rootfs and initramfs. Linux Kernel Documentation. `http://www.kernel.org/doc/Documentation/filesystems/ramfs-rootfs-initramfs.txt`, October 2005.

[83] Erwin Laure, A. Edlund, F. Pacini, P. Buncic, M. Barroso, A. Di Meglio, F. Prelz, A. Frohner, O. Mulmo, A. Krenek, et al. Programming the Grid with gLite. Technical report, 2006.

[84] lessfs authors. lessfs – Open Source Data Deduplication for Less. `http://www.lessfs.com/`, April 2012.

[85] The LibVMI Project. LibVMI – Virtual Machine Introspection. Fast, Portable, Simple. `http://libvmi.com/`, 2015.

[86] Anthony Liguori and Eric Van Hensbergen. Experiences with Content Addressable Storage and Virtual Disks. In *Proceedings of the 1st USENIX Workshop on I/O Virtualization (WIOV '08)*, pages 5–10, Berkeley, CA, USA, 2008. USENIX Association.

[87] Terrence V. Lillard, Clint P. Garrison, Craig A. Schiller, and James Steele. The Future of Cloud Computing. In *Digital Forensics for Network, Internet, and Cloud Computing*, pages 319 – 339. Syngress, Boston, 2010.

[88] LINBIT HA-Solutions GmbH. DRBD - Software Development for High Availability Clusters. `http://www.drbd.org/`, 2011.

[89] The Linux Foundation. Linux Standard Base. `http://refspecs.linux-foundation.org/lsb.shtml`, May, 2015.

[90] The Linux man-pages project. Linux Programmer's Manual - Linux system calls. `http://man7.org/linux/man-pages/man2/syscalls.2.html`, February 2013.

[91] llfuse authors. Python-LLFUSE. `http://pythonhosted.org/llfuse/`, 2015.

[92] Yingwei Luo, Binbin Zhang, Xiaolin Wang, Zhenlin Wang, Yifeng Sun, and Haogang Chen. Live and Incremental Whole-system Migration of Virtual Machines Using Block-Bitmap. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing (CLUSTER 2008)*, pages 99–106. IEEE Press, 2008.

[93] Ravi K. Madduri, Cynthia S. Hood, and William E. Allcock. Reliable File Transfer in Grid Environments. In *Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN '02)*, pages 737–738, Washington, DC, USA, Nov 2002. IEEE Computer Society.

[94] Markus Mathes, Steffen Heinzl, Roland Schwarzkopf, and Bernd Freisleben. F&L-Grid: Eine generische Backup und Recovery Infrastruktur für das D-Grid. In *Tagungsband des 2. DFN-Forum Kommunikationstechnologien*, pages 55–67. Gesellschaft für Informatik (GI), 2009.

[95] Markus Mathes, Roland Schwarzkopf, Tim Dörnemann, Steffen Heinzl, and Bernd Freisleben. Orchestration of Time-Constrained BPEL4WS Workflows. In *Proceedings of the 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4. IEEE Computer Society Press, 2008.

[96] Markus Mathes, Roland Schwarzkopf, Tim Dörnemann, Steffen Heinzl, and Bernd Freisleben. Composition of Time-Constrained BPEL4WS Workflows using the TiCS Modeler. In *Proceedings of the 13th IFAC Symposium on Information Control Problems in Manufacturing (INCOM)*, pages 892–897. Elsevier, 2009.

[97] Markus Mathes, Christoph Stoidner, Roland Schwarzkopf, Steffen Heinzl, Tim Dörnemann, Helmut Dohmann, and Bernd Freisleben. Time-Constrained Services: a Framework for using Real-Time Web Services in Industrial Automation. *Service Oriented Computing and Applications*, 3(4):239–262, 2009.

[98] Heinz Mauelshagen and Matthew O'Keefe. The Linux Logical Volume Manager. *redhat Magazine*, (9), July 2005.

[99] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: Virtual Disks for Virtual Machines. *ACM SIGOPS Operating System Review*, 42(4):41–54, April 2008.

[100] Dutch T. Meyer and William J. Bolosky. A Study of Practical Deduplication. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST '11)*, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[101] Gal Motika and Shlomo Weiss. Virtio Network Paravirtualization Driver: Implementation and Performance of a De-facto Standard. *Computer Standards & Interfaces*, 34(1):36 – 47, 2012.

[102] Ian Murdock. How Package Management Changed Everything. `http://ianmurdock.com/solaris/how-package-management-changed-everything/`, 2007.

[103] Partho Nath, Michael Kozuch, David R. O'Hallaron, Jan Harkes, Mahadev Satyanarayanan, Niraj Tolia, and Matt Toups. Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference (ATC '06)*, pages 71–84, Berkeley, CA, USA, 2006. USENIX Association.

[104] National Security Agency. Security-Enhanced Linux. `http://www.nsa.gov/research/selinux/`, 2009.

[105] Chun-Ho Ng, Mingcao Ma, Tsz-Yeung Wong, Patrick P. C. Lee, and John C. S. Lui. Live Deduplication Storage of Virtual Machine Images in an Open-source Cloud. In *Proceedings of the 12th International Middleware Conference (Middleware '11)*, pages 80–99, Laxenburg, Austria, 2011. International Federation for Information Processing.

[106] Bogdan Nicolae, Franck Cappello, and Gabriel Antoniu. Optimizing Multi-deployment on Clouds by Means of Self-adaptive Prefetching. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Proceedings of the Euro-Par 2011: Parallel Processing Workshops*, volume 6852 of *Lecture Notes in Computer Science*, pages 503–513. Springer Berlin Heidelberg, 2011.

[107] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09)*, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[108] Junjiro R. Okajima. aufs3 – Advanced Multi Layered Unification Filesystem Version 3.x. `http://aufs.sourceforge.net/`, November 2012.

[109] The Open Group. POSIX.1-2008 / IEEE Std 1003.1-2008. `http://www.unix.org/version4/ieee_std.html`, 2008.

[110] Open Source Security, Inc. grsecurity. `http://grsecurity.net/`, 2015.

[111] The OpenStack Project. OpenStack Open Source Cloud Computing Software. `http://www.openstack.org/`, August 2015.

[112] OpenVAS authors. The Open Vulnerability Assessment System (OpenVAS). `http://www.openvas.org/`, 2012.

[113] Oracle Corporation. Oracle Solaris ZFS. `http://www.oracle.com/us/products/servers-storage/solaris/solaris-zfs-ds-067320.pdf`, 2011.

[114] Oracle Corporation. VirtualBox. `http://www.virtualbox.org`, 2015.

[115] The oVirt Project. oVirt - Open Your Virtual Datacenter. `http://www.ovirt.org`, 2015.

[116] Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation (NSDI '06)*, pages 26–26, Berkeley, CA, USA, 2006. USENIX Association.

[117] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[118] Satya Popuri. A Tour of the Mini-OS Kernel. `http://www.cs.uic.edu/~spopuri/minios.html`, 2012.

[119] Shaya Potter and Jason Nieh. AutoPod: Unscheduled System Updates with Zero Data Loss. In *Proceedings of the International Conference on Autonomic Computing*, pages 367–368, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[120] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing Applications. In *In Proceedings of the USENIX Technical Annual Conference (ATC '01)*, pages 119–126, Berkeley, CA, USA, 2001. USENIX Association.

[121] PT-GRID Project. Modellierung und Simulation von plasmatechnologischen Anwendungen im Grid. `http://www.pt-grid.de/`, 2012.

[122] 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*, volume 2, pages 349–362, Ottawa, Canada, July 2006.

[123] Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Data Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[124] Michael O. Rabin. *Fingerprinting by Random Polynomials.* Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, 1981.

[125] Kaveh Razavi and Thilo Kielmann. Scalable Virtual Machine Deployment Using VM Image Caches. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*, pages 65:1–65:12, New York, NY, USA, 2013. ACM.

[126] Red Hat, Inc. Software Management Guide. `http://docs.fedoraproject.org/en-US/Fedora/18/html/System_Administrators_Guide/`, 2013.

[127] Joshua Reich, Oren Laadan, Eli Brosh, Alex Sherman, Vishal Misra, Jason Nieh, and Dan Rubenstein. VMTorrent: Scalable P2P Virtual Machine Streaming. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)*, pages 289–300, New York, NY, USA, 2012. ACM.

[128] Darrell Reimer, Arun Thomas, Glenn Ammons, Todd Mummert, Bowen Alpern, and Vasanth Bala. Opening Black Boxes: Using Semantic Information to Combat Virtual Machine Image Sprawl. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)*, pages 111–120, New York, NY, USA, 2008. ACM.

[129] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th Conference*

on USENIX Security Symposium (SSYM '00), pages 10–10, Berkeley, CA, USA, 2000. USENIX Association.

[130] Alessandro Rubini. The "Virtual File System" in Linux. *Linux Journal*, (37), May 1997.

[131] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual Appliances for Deploying and Maintaining Software. In *Proceedings of the 17th USENIX Conference on System Administration (LISA '03)*, pages 181–194, Berkeley, CA, USA, 2003. USENIX Association.

[132] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. *ACM SIGOPS Operating System Review*, 36(SI):377–390, December 2002.

[133] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, October 2009.

[134] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, David, and C. Steere. Coda: A Highly available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39:447–459, 1990.

[135] The SAX Project. About SAX. `http://www.saxproject.org`, April 2004.

[136] Matthias Schmidt, Sascha Fahl, Roland Schwarzkopf, and Bernd Freisleben. TrustBox: A Security Architecture for Preventing Data Breaches. In *Proceedings of the 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '11)*, pages 635–639, Washington, DC, USA, Feb 2011. IEEE Computer Society.

[137] Matthias Schmidt, Niels Fallenbeck, Kay Dörnemann, Roland Schwarzkopf, Tobias Pontz, Manfred Grauer, and Bernd Freisleben. Aufbau einer virtualisierten Cluster-Umgebung. In Oliver Hinz, Roman Beck, Bernd Skiera, and Wolfgang König, editors, *Grid Computing in der Finanzindustrie*, pages 119–131. Books on Demand, Norderstedt, 2009.

[138] Matthias Schmidt, Niels Fallenbeck, Matthew Smith, and Bernd Freisleben. Efficient Distribution of Virtual Machines for Cloud Computing. In *Proceedings of the Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, pages 567–574. IEEE Computer Society, 2010.

[139] Roland Schwarzkopf, Thomas Gebhardt, Matthias Schmidt, and Bernd Freisleben. Management virtueller Maschinen in Linux High Performance Clustern. In *Forschungsbericht Hochleistungsrechnen in Hessen 2015*, page 106, 2015.

[140] Roland Schwarzkopf, Markus Mathes, Steffen Heinzl, Bernd Freisleben, and Helmut Dohmann. Java RMI versus .NET Remoting - Architectural Comparison and Performance Evaluation. In *Proceedings of the 7th International Conference on Networking (ICN)*, pages 398–407. IEEE Computer Society Press, 2008.

[141] Roland Schwarzkopf, Matthias Schmidt, Niels Fallenbeck, and Bernd Freisleben. Multi-Layered Virtual Machines for Security Updates in Grid Environments. In *Proceedings of 35th Euromicro Conference on Internet Technologies, Quality of Service and Applications (ITQSA)*, pages 563–570. IEEE Press, 2009.

[142] Roland Schwarzkopf, Matthias Schmidt, Mathias Rüdiger, and Bernd Freisleben. Efficient Storage of Virtual Machine Images. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing (ScienceCloud '12)*, pages 51–60, New York, NY, USA, 2012. ACM.

[143] Roland Schwarzkopf, Matthias Schmidt, Christian Strack, and Bernd Freisleben. Checking Running and Dormant Virtual Machines for the Necessity of Security Updates in Cloud Environments. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 239–246. IEEE Press, 2011.

[144] Roland Schwarzkopf, Matthias Schmidt, Christian Strack, Simon Martin, and Bernd Freisleben. Increasing Virtual Machine Security in Cloud Environments. *Journal of Cloud Computing: Advances, Systems and Applications*, 1(1), 2012.

[145] sdfs authors. Opendedup SDFS. `http://opendedup.org/`, January 2013.

[146] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck. RFC 3530. Network File System (NFS) version 4 Protocol. `http://tools.ietf.org/html/rfc3530`, April 2003.

[147] Matthew Smith, Matthias Schmidt, Niels Fallenbeck, Tim Dörnemann, Christian Schridde, and Bernd Freisleben. Secure On-demand Grid Computing. *Journal of Future Generation Computer Systems*, 25(3):315–325, March 2009.

[148] Softpanorama - Open Source Software Educational Society. NFS Performance Tuning. `http://www.softpanorama.org/Net/Application_layer/NFS/nfs_performance_tuning.shtml`, August 2009.

[149] Karen Sollins. RFC 1350. The TFTP Protocol (Revision 2). `http://tools.ietf.org/html/rfc1350`, July 1992.

[150] Borja Sotomayor, Kate Keahey, and Ian Foster. Overhead Matters: A Model for Virtual Resource Management. In *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing (VTDC '06)*, pages 1–5, Washington, DC, USA, 2006. IEEE Computer Society.

[151] A. Streit, D. Erwin, Th. Lippert, D. Mallmann, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and Ph. Wieder. Unicore — From Project Results to Production Grids. In Lucio Grandinetti, editor, *Grid Computing The New Frontier of High Performance Computing*, volume 14 of *Advances in Parallel Computing*, pages 357 – 376. North-Holland, 2005.

[152] Chunqiang Tang. FVD: A High-Performance Virtual Machine Image Format for Cloud. In *Proceedings of the USENIX Annual Technical Conference (ATC '11)*, pages 1–6, Berkeley, CA, USA, 2011. USENIX Association.

[153] Pedro Henriques dos Santos Teixeira, Ricardo Gomes Clemente, Ronald Andreu Kaiser, and Denis Almeida Vieira, Jr. HOLMES: An Event-driven Solution to Monitor Data Centers through Continuous Queries and Machine Learning. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems (DEBS '10)*, pages 216–221, New York, NY, USA, 2010. ACM.

[154] Tenable Network Security. Nessus Security Scanner. `http://www.nessus.org/products/nessus`, 2012.

[155] TIMaCS Project. TIMaCS – Tools for Intelligent Systems Management of Very Large Computing Systems. `http://www.timacs.de`, 2015.

[156] TurnKey Linux. All virtual appliances | TurnKey GNU/Linux. `http://www.turnkeylinux.org/all`, July, 2015.

[157] Unionfs authors. Unionfs: A Stackable Unification File System. `http://www.filesystems.org/project-unionfs.html`, August 2012.

[158] virt-manager Project. Manage Virtual Machines with virt-manager. `http://www.virt-manager.org`, 2015.

[159] VMware, Inc. VMware Products. `http://www.vmware.com/products/`, 2015.

[160] VMware, Inc. Understanding Full Virtualization, Paravirtualization, and Hardware Assist. `http://www.vmware.com/resources/techresources/1008`, November 2007.

[161] Werner Vogel. Amazon DynamoDB – a Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications. `http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html`, 2012.

[162] Eugen Volk, Jochen Buchholz, Stefan Wesner, Daniela Koudela, Matthias Schmidt, Niels Fallenbeck, Roland Schwarzkopf, Bernd Freisleben, Götz Isenmann, Jürgen Schwitalla, Marc Lohrer, Erich Focht, and Andreas Jeutter. Towards Intelligent Management of Very Large Computing Systems. In Christian Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum, editors, *Competence in High Performance Computing 2010*, pages 191–204. Springer Berlin Heidelberg, 2012.

[163] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation. In *Proceedings of the 1st International Conference on Cloud Computing (CloudCom '09)*, pages 254–265, Berlin, Heidelberg, 2009. Springer-Verlag.

[164] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In *Proceedings of the 25th Annual International Conference on Advances in Cryptology (CRYPTO '05)*, pages 17–36, Berlin, Heidelberg, 2005. Springer-Verlag.

[165] WarCat team. Debian OpenSSL Predictable PRNG Bruteforce SSH Exploit. `http://www.exploit-db.com/exploits/5720`, June 2008.

[166] Andrew Warfield, Russ Ross, Keir Fraser, Christian Limpach, and Steven Hand. Parallax: Managing Storage for a Million Machines. In *Proceedings of the 10th*

Conference on Hot Topics in Operating Systems (HOTOS'05), pages 4–4, Berkeley, CA, USA, 2005. USENIX Association.

[167] Jinpeng Wei, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, and Peng Ning. Managing Security of Virtual Machine Images in a Cloud Environment. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security (CCSW '09)*, pages 91–96, New York, NY, USA, 2009. ACM.

[168] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. *ACM SIGOPS Operating Systems Review*, 36(SI):195–209, December 2002.

[169] David Winer. XML-RPC Specification. `http://www.xml-rpc.com/spec`, Jun 2003.

[170] David Isaac Wolinsky, Abhishek Agrawal, P. Oscar Boykin, Justin R. Davis, Arijit Ganguly, Vladimir Paramygin, Y. Peter Sheng, and Renato J. Figueiredo. On the Design of Virtual Machine Sandboxes for Distributed Computing in Wide-area Overlays of Virtual Workstations. In *Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing (VTDC '06)*, page 8, Washington, DC, USA, 2006. IEEE Computer Society.

[171] The World Wide Web Consortium (W3C). XML Path Language (XPath) 3.1. `http://www.w3.org/TR/xpath-3/`, December 2014.

[172] The World Wide Web Consortium (W3C). Document Object Model (DOM). `http://www.w3.org/DOM/`, January 2015.

[173] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad Nayyer Zubair. Versatility and Unix Semantics in Namespace Unification. *ACM Transactions on Storage (TOS)*, 2(1):74–105, February 2006.

[174] The Xen Project. Xen Project Software Overview. `http://wiki.xenproject.org/wiki/Xen_Project_Software_Overview`, 2015.

[175] The Xen Project. pyGRUB. `http://wiki.xenproject.org/wiki/PyGrub`, April 2010.

[176] The Xen Project. Xen PV-GRUB. `http://wiki.xenproject.org/wiki/PvGrub`, May 2010.

[177] Jun Yoon and Wontae Sim. Implementation of the Automated Network Vulnerability Assessment Framework. In *Proceedings of the 4th International Conference on Innovations in Information Technology (IIT '07)*, pages 153 –157. IEEE Press, Nov 2007.

[178] Zhen Zhou, Shuyu Chen, Mingwei Lin, Guiping Wang, and Qian Yang. Minimizing Average Startup Latency of VMs by Clustering-based Template Caching Scheme in an IaaS System. *International Journal of u-and e-Service, Science and Technology*, 6(6):133–146, 2013.

Curriculum Vitae

Ausbildung

10/2007 – 12/2015	Philipps-Universität Marburg Promotionsstudium Informatik Promotion zum Dr. rer. nat. (Gesamtnote "sehr gut")
09/2006 – 09/2007	Philipps-Universität Marburg Studiengang Informatik Ohne Abschluss (Erfüllung von Auflagen des Promotionsausschusses)
09/2002 – 08/2006	Hochschule Fulda Studiengang Angewandte Informatik Abschluss als Diplom-Informatiker (FH) mit Auszeichnung (Note 1,0)
08/1998 – 08/2001	Freiherr-vom-Stein-Schule, Fulda Abschluss mit dem Zeugnis der Allgemeinen Hochschulreife

Berufserfahrung

seit 10/2007	Philipps-Universität Marburg Wissenschaftlicher Mitarbeiter
10/2007 – 04/2010	Universität Siegen Wissenschaftlicher Mitarbeiter
02/2006 – 07/2006	Kinzig-Schule Schlüchtern Vertretungslehrer in einem ergänzenden Grundkurs Informatik
10/2005 – 09/2007	Hochschule Fulda Studentische Hilfskraft in Lehre und Forschung (mit Unterbrechungen)