Virtual Appliance Size Optimization with Active Fault Injection

Gabor Kecskemeti, Gabor Terstyanszky and Peter Kacsuk

Abstract—Virtual appliances store the required information to instantiate a functional virtual machine on Infrastructure as a Service (IaaS) cloud systems. Large appliance size obstructs IaaS systems to deliver dynamic and scalable infrastructures according to their promise. To overcome this issue, this article offers a novel technique for virtual appliance developers to publish appliances for the dynamic environments of IaaS systems. Our solution achieves faster virtual machine instantiation by reducing the appliance size while maintaining its key functionality. The new virtual appliance optimization algorithm identifies the removable parts of the appliance. Then, it applies active fault injection to remove the identified parts. Afterwards, our solution assesses the functionality of the reduced virtual appliance by applying the – appliance developer provided – validation algorithms. We also introduce a technique to parallelize the fault injection and validation phases of the algorithm. Finally, the prototype implementation of the algorithm is discussed to demonstrate the efficiency of the proposed algorithm through the optimization of two well-known virtual appliances. Results show that the algorithm significantly decreased virtual machine instantiation time and increased dynamism in IaaS systems.

Index Terms—Virtual appliance, Optimization, Cloud Computing, IaaS

1 Introduction

Infrastructure as a service (IaaS – [1], [2]) cloud systems promise to provide on demand and scalable infrastructures. The scalability of this new kind of infrastructure is provided through virtualization [3]. Even the most basic IaaS systems offer service interfaces to create and manage virtual machines (VMs) hosted by various virtual machine monitors [4], [5], [6]. From the earliest commercial IaaS systems – like Amazon EC2 – these systems are marketed as an approach to dynamically extend service infrastructures. Users prepare virtual appliances (VAs – [7]) hosting the dynamic components of their service-based applications. These appliances store the necessary information to instantiate a functional virtual machine (e.g. an operating system, the dependencies and the code of the dynamic component itself).

One of the most frequently referred scenarios of commercial IaaS providers offers a solution to handle the peak demand periods of service-based applications [8], [9], [10]. The highly dynamic [11], [12] nature of these applications require IaaS users to instantiate their virtual appliances on demand within a minimal amount of time. However, the virtual appliance instantiation time is mainly dependent on the size of the appliance. Therefore, appliance developers should create virtual appliances with the smallest size while they still maintain the key functionality of the appliance. The manual creation of such appliances requires expertise and time not available for most IaaS users. Consequently, highly dynamic service environments require techniques for creating virtual appliances.

Already existing automated appliance creation techniques (e.g. [13]) utilize the dependencies between the various software components of the future virtual appliance. These techniques require the appliance developer to define the dependencies and requirements of its dynamic components before creating the appliance itself. Then, they construct the virtual appliance according to the previous definitions. As a result, these techniques rely on the virtual appliance developer's skills of preparing optimally sized appliances for dynamic service environments. This research reduces the demands on the appliance developers by requiring them to only define the intended usage scenarios (referred as the key functionality) of dynamic components.

Active fault injection is a well-discussed topic for determining the fault tolerance of various software systems [14], [15], [16], [17]. This article introduces the concept of active fault injection to a new domain: virtual appliance size minimization. We propose to inject faults to virtual appliances by removing their parts while they are executed in virtual machines. The article offers a fault injection technique that is independent from the granularity of the virtual appliance parts (e.g. they can be software packages or files). This technique first identifies the smallest individually handled parts of the virtual appliance. Afterwards, the technique identifies the parts more favorable for removal. After their removal, the reduced virtual appliance is validated against the intended usage scenarios specified by the appliance developer.

- *G. Kecskemeti and P. Kacsuk are with the Laboratory of Parallel and Distributed Systems at Computer and Automation Institute of the Hungarian Academy of Sciences, Kende u. 13-17, Budapest 1111, Hungary E-mail: {kecskemeti,kacsuk}@sztaki.hu*
- *G. Terstyanszky is with the Centre of Parallel Computing at University of Westminster, 309 Regent Street, London W1B 2UW, United Kingdom E-mail: G.Z.Terstyanszky@westminster.ac.uk*

The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

Digital Object Indentifier 10.1109/TPDS.2011.309

Finally, the article discusses an approach to parallelize the removal and validation tasks of our technique.

This new technique is based on the assumption that the appliance developer intends to create single purpose virtual appliances. These appliances have a well defined and single functionality (e.g. provide a specific website). We assume that the appliance developer is capable to describe the purpose of the appliance by providing validation algorithms for the key functionality. In contrast, generic appliances are designed to be customizable by third parties (e.g. LAMP – Linux, Apache, MySQL, php – appliances that allow their arbitrary customization after deployment). As the developers cannot clearly specify the later use of their appliances they cannot define the validation algorithms. Consequently, the approach is not applicable to generic virtual appliances but only to a subclass of virtual appliances paired with the definition of their validator algorithms. The research results are utilized as part of an Automated Virtual appliance creation Service that was introduced and discussed in [18], [19].

This article discusses our implementation that is instantly applicable to IaaS systems similar to Amazon Elastic Compute Cloud (*EC2* – [20]). Then, we evaluate the proposed technique through experiments executed on a Eucalyptus [21] based testbed. The experiments first present the significant virtual appliance size reduction (e.g. more than 90%) achievable with our technique. Second, the article reveals how the optimization time is reduced by refining the proposed technique reaching less than 2-hour optimization time with a 32 CPU based cluster on our test VAs (Apache and SSH).

This paper is organized as follows. Section 2 provides an overview about the related works. Then, Section 3 highlights the architecture that is proposed to include the optimization approach described in the article. Later, in Section 4, we discuss the basics of the optimization algorithm. Then, Section 5 reveals the details and the decisions we made for the first implementation. Finally, in Section 6 we measure the effectiveness of the implemented optimization algorithm.

2 Related Works

Virtual appliance distribution can be optimized by either optimizing the delivery path of the appliance (e.g. by caching or replicating some of the appliance contents for faster, multi-sourced delivery [19], [22], [23]) or by reducing its size. Size reduction results immediate effects on appliance instantiation time even without delivery path changes. This article is focused on the second approach, therefore this related works section is only focusing on the appliance size optimization approaches.

The discussed approaches can be classified based on their input requirements. The *pre-optimizing* approach requires the appliance developer to provide the application and its known dependencies that should be offered by the appliance. In contrast, the *post-optimizing* approach uses already existing but non-optimized appliances.

With *pre-optimizing* algorithms the dependencies of the user applications are prepared as reusable virtual appliance components. The appliance developers select from these components so that they can form the base of the user application. These algorithms then form the virtual appliance with the selected reusable components and the application itself. RBuilder [13] applies this algorithm with an extension that supports creating custom virtual appliances by building from the source code.

The extreme case of pre-optimizing algorithms is the minimalist pre-optimizing approach that offers optimized virtual appliances with known software environments. To support this approach several OS and reusable application vendors offer the minimalist version of their product packaged together with their just-enough operating system (JeOS – [24]) in virtual appliances [25], [26], [27], [28]. For example, there are several virtual appliances available prepared to host a simple LAMP project. However, this approach requires the appliance developer to manually install its application to a suitable optimized virtual appliance. The advantage of these algorithms is the fast creation of the appliances but at the price that the developer has to trust the optimization attempt of the used virtual appliance's vendor. If the appliance is not well optimized, or the vendor offers a generic appliance for all uses then the descendant virtual appliances cannot be optimal without further efforts.

Other pre-optimizing algorithms determine dependencies within the virtual appliance by using its source code. Software clone [29] and dependency [30] detection techniques identify all of the required underlying software components by analyzing the sources. Once the dependencies are detected these algorithms leave only those components that are required for serving the key functionality of the virtual appliance. Optimizing a virtual appliance with these techniques require the source code of all the software encapsulated within the appliance. Thus they also need to analyze the underlying systems (e.g. the operating system) of the application. Unfortunately this last requirement renders these techniques unfeasible in most cases.

The most widely used *post-optimizing algorithms* are optimizing the free space in the disk images of the virtual appliance. If virtual appliances are created from previously used software systems then their disk images contain their available free space fragmented throughout the entire image. Before publication, virtual appliances are usually compressed for easier transfer. However, the fragmented free space is harder to compress. Therefore, these post optimizing algorithms analyze the available free space and first they fill them with easily compressible data. Next, they offer their users the option to defragment the disk images. As a result, these disk images can be shrunk so they are not only more compressible but they do not even store the free space in the disk image if they are not required. The advantage of this algorithm, that it can operate on any virtual appliance so long it can understand and use its file system. This

technique is utilized by [31].

Our investigations show that the field of post-optimizing algorithms is less developed, even though they could bring remedy for the vast amounts of already existing virtual appliances that are currently unusable in dynamic service environments. Consequently, the size optimization technique discussed in this article can also be classified as a post-optimizing algorithm.

3 The Automatic Service Deployment Architecture

In our previous work [18] we introduced the *Automatic Service Deployment (ASD)* architecture as seen in Fig. 1. The architecture provides a framework to our recent and future works. This section gives a short overview of the architecture and contextualizes current research results.

Fig. 1 presents the service deployment architecture that supports various tasks of the virtual appliance-based deployment and also depicts several issues that the current solutions do not aim at. First, the architecture offers a solution for initial virtual appliance creation. Virtual appliances are acquired and managed by the *Automated Virtual appliance creation Service* [19]. In the article only the following virtual appliance preparation tasks were solved: (i) extracting a virtual appliance from a running system, (ii) transformation of the appliance between various virtual machine image formats, and (iii) uploading the virtual appliance to a repository. Through the functionality of the *optimization facility*, this article extends these tasks with the size optimization of the previously extracted virtual appliances.

Second, the architecture also supports developers to create appliances more suitable for highly dynamic environments via *minimal manageable virtual appliances* [32] that they can use as the base of their appliance. This appliance provides *management interfaces* offering package installation, configuration and removal operations.

Third, the architecture defines *active repositories* with the following automatic entry management functionality: (i) appliance decomposition, (ii) package merging, (iii) destruction and (iv) partial replication. Therefore, these repositories have the ability to identify common virtual appliance parts and optimize their storage.

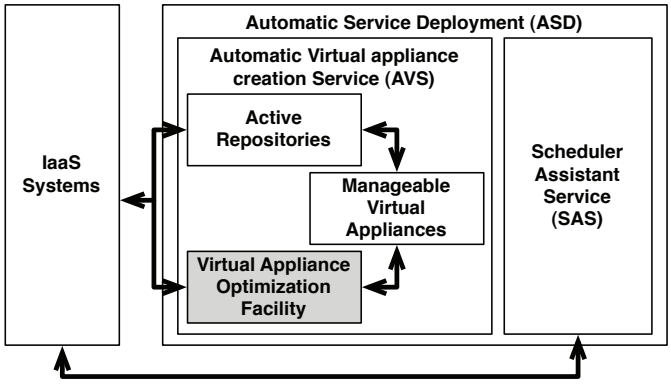


Fig. 1. Appliance optimization within ASD

Finally, the architecture also offers decision-making support for schedulers and other high level entities that make deployment related decisions in a dynamic service ecosystem. Service deployment can be initiated by higher-level components (such as service schedulers or service composition engines), the *Scheduler Assistance Service (SAS)* aids their deployment decisions by offering interfaces to identify deployable services and rank sites that can host them. The research issues and design considerations of the SAS are discussed in [33].

4 Virtual Appliance Optimization Principles

As the first step towards the optimization algorithm, this section identifies the time reduction options for virtual appliance instantiation. We have defined the instantiation as a composite task of three major steps: (i) download the appliance, (ii) initialize the VM and *start up* its operating system and finally, (iii) activate the service in the VM. Fig. 10 presents the average execution time of these three tasks for two typical virtual appliances described in Section 6.3.

It can be observed in Fig. 10 that instantiation time mainly depends on the download time of the virtual appliance. This time can be optimized in two ways: (i) storing the virtual appliance in a repository with the smallest latency and largest transfer rate towards its executor host and (ii) minimizing the size of the virtual appliance while still maintaining its key functionality. The first option is only viable with IaaS systems that support multiple repositories (this is not the case with Amazon EC2 [20] or Eucalyptus [21]), therefore to support wider range of IaaS systems we only focus on the size minimization in this article. Throughout this article, we did not make any assumptions on the IaaS behavior; therefore, this approach is applicable to any IaaS system.

4.1 The Virtual Appliance Optimization Facility

Before detailing the features of our size optimization technique, we define the constraints of the optimization facility and system. The *optimization facility* incorporates the algorithms described in this article. It resides on a single host of the *optimization system* (φ). The optimization system incorporates multiple hosts ($h_x \in \varphi$) within a single administrative domain. These hosts also implement an IaaS system offering the required virtual machine management functionality for the facility.

The main task of the optimization facility is to solve the $optimize : P \times \mathcal{C} \rightarrow P'$ function. Where the set P refers to all virtual appliances in the optimization system, and $\mathcal{C}$ defines the possible *completion conditions* that can limit the optimization according to the appliance developer. At the end of the optimization operation, the $p'_\sigma := optimize(p_\sigma, c)$ function provides a smaller sized

version (p'_σ) of the original virtual appliance (p_σ) and publish it among the other virtual appliances: $P' := P \cup \{p'_\sigma\}$. If the optimization function receives an empty completion condition set ($c = \emptyset$) then the optimization is executed until p'_σ becomes *optimally sized* – see Eq. (4).

Active fault injection uses fault injection techniques that generate hardware and software level faults to test the fault tolerant behavior of software. However, for virtual appliance optimization, we do not test for fault tolerant behavior. Instead, fault injection identifies those parts of the p_σ that are not needed for its key functionality.

First, we define the faults that can be injected in order to achieve size reduction. Having a virtualized environment enables the simulation of both software and hardware level faults. Software level faults could arise on the file system – e.g. simulating the corruption of the file system by removing a file or some of its parts. Hardware faults could occur in the memory or the disk subsystem. Simulating hardware faults need changes in virtual machine monitors. This requirement seriously reduces the adoptability of an optimization technique; thus, this article only considers software fault injection.

4.2 Appliance Contents Removal

This section describes the basic algorithm of the optimization procedure that is split into four tasks as seen in Fig. 2. The first task is the *selection* of the virtual appliance's removable parts. The selection task is the most complex and critical task of the optimization approach. This task analyzes the package ($p_\sigma \in P$) of the virtual appliance and proposes how to partition the appliance.

The selection task also assigns a weight value to appliance parts. The highest weighted parts are temporarily erased from the appliance by the *removal* task. The third task is the *validation* of the modified appliance using validation algorithms provided by the appliance developer. This task decides whether the erased items should be permanently removed from the VA. Later, we refer to the triplet of selection, removal and validation as the *optimization iteration* (see the bold arrows in Fig. 2).

The last task of the algorithm decides whether the system should initiate further optimization iterations. The decision is based on the appliance developer provided *completion conditions*. If the conditions are met the algorithm publishes the optimized appliance. Otherwise, the algorithm proceeds with the next iteration. These four tasks are further outlined in the next sub-sections.

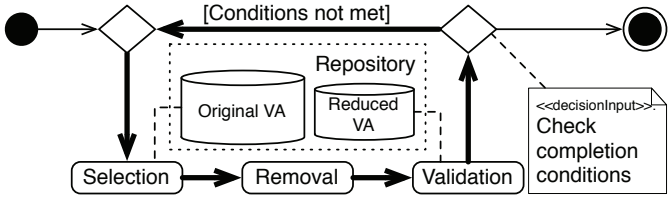


Fig. 2. Basic appliance optimization technique

4.2.1 Selection

As it was discussed previously, the main task of the selection is to identify parts to be removed. From the selection point of view the granularity of the parts in the VA is not important. However, the selection algorithm should use the same appliance part granularity that the removal task uses. Virtual appliance parts can range from single bytes, sectors, file contents, files to even directories or software packages.

The identification of the different parts and their metadata is called *itemization*. *Items* ($i \in I$) are the internal representation of the virtual appliance parts with metadata. Different itemization techniques use different kinds of virtual appliance parts as items. The set of I represents all possible items created with a particular itemization technique. Items are the smallest entities handled by the selection and removal algorithms. The entire item set of an appliance is represented with the function $it : P \to I$.

If multiple itemization techniques are available, then the optimization is executed in several phases. In the first phase, the facility removes and validates the item set of the current itemization technique. If there are no more items to validate and the completion conditions allow, the facility initiates a new phase using an itemization technique with smaller granularity.

For example, if the first applied itemization technique uses software package managers (e.g. the debian package manager [34] – `dpkg`), then the facility first removes all software packages not related to the key functionality of the appliance. The facility switches to smaller granularity when it cannot purge more packages from the VA with the package manager. Consequently, during later executed phases, the facility could even erase the software package manager itself if it is not required for the key functionality of the appliance.

The second subtask of selection categorizes the various items according to the following list: (i) the *core items* – i_c –, (ii) the *volatile items* – i_v – and finally, (iii) the *fuzzy items* – i_f. Those items that the selection algorithm does not have any prior knowledge about are called volatile and are primarily exposed to active fault injection by the optimization facility. In contrast to volatile items, the core and fuzzy items are identified by the past knowledge of the selection task. The core items cannot be removed from an appliance under any circumstances (e.g. the `init` application in SystemV compatible UNIX systems); these items are predefined in the knowledge base of the selection task. Initially no fuzzy items are defined. They are identified as those items that – according to the knowledge base – were repeatedly validated unsuccessfully in prior optimization operations. If the developer specified completion conditions could not be reached, then the facility starts removing and validating fuzzy items before offering the optimized appliance.

Weight functions ($w : I \times P \to V$) assign weight values ($V = \{v \in \mathbb{R} : (0 \leq v < 1)\}$) for each item of the virtual appliance under optimization ($i \in it(p_\sigma)$).

Items with higher weight values are more likely to be removed. As a result, for any combination of core, fuzzy and volatile items the following statement is always true:

$$0 = w(i_c, p) < w(i_f, p) < w(i_v, p) \quad (1)$$

Weight functions utilize metadata available about the items. As a result, methods for collecting new metadata have to be specified along with the new weighting algorithms. Therefore, details of the weight functions and the collected metadata are discussed in Section 5.1.

The optimization facility decides on the use of the various weight functions based on the time and cost constraints specified in the completion conditions. The facility can even decide whether to use a single or multiple weight functions throughout the optimization process. Alternatively, different weight functions can be used during the different stages of the optimization process. This strategy ensures that the more expensive and more precise weight calculations used only if they would result faster size reduction.

4.2.2 Removal

The *removal task* sorts the items according to their weights and removes the highest weighted item ($i_{hw} \in I$) from the original virtual appliance (p_σ).

$$i_{hw} := i \in it(p_\sigma) : \left(w(i, p_\sigma) = \max_{j \in it(p_\sigma)} w(j, p_\sigma)\right) \quad (2)$$

$$it(p_{red}) := it(p_\sigma) \backslash \{i_{hw}\} \quad (3)$$

Where p_{red} represents the newly created *reduced virtual appliance* that no longer includes i_{hw}. For the removal operation the optimization facility has to understand the structure of the appliance and the item type of the used itemization technique to be able to remove the highest weighted item. For instance, the facility has to handle the file system of the VA in case of file based itemization. We have identified two basic techniques for removal: (i) pre-execution and (ii) during execution.

First, *pre-execution removal* operates on the items of the virtual appliance while it is not running. This technique first attaches the disk images of the original virtual appliance to the optimization facility's host, and removes the item with the highest weight. Then to allow the validation of the new appliance, it is initiated in a virtual machine. However, IaaS systems similar to Amazon EC2 only initiate virtual machines with virtual appliances stored in their repositories. This requirement forces the optimization facility to upload the reduced virtual appliance to the repository of the IaaS system before validation. Thus, the facility applies this removal technique if the optimization system uses an IaaS system capable of instantiating virtual machines with appliances from external sources (e.g., from the optimization facility).

Second, if the virtual appliance under optimization offers management interfaces (e.g. [35], [36]) then it enables a new removal technique, called *removal during execution*. This approach requires the original virtual appliance instantiated in a virtual machine before the removal operation takes place. Then, the highest weighted items are removed using the management capabilities of the virtual machine. Consequently, the reduced virtual appliance (p_{red}) is created by altering the virtual machine of the original appliance. However, the functionality of the appliance could remain intact even after the management interfaces remove i_{hw} from the virtual machine because its memory still holds the removed item. Thus, the VM is restarted to erase its memory before validation. The restart cleans up the claimed address space of the VM, thus avoids security issues unclaimed memory could cause – beyond this scope we did not consider further security issues. The validation automatically fails if the restart fails; otherwise it is executed on the running virtual machine. This approach is the only practical solution in EC2 like IaaS systems, because it does not require the repeated upload of the reduced virtual appliances. This article applies the management interfaces for active fault injection only, the detailed definition of these interfaces and their further usage options are discussed in [32].

Based on the availability of the management interfaces the optimization facility automatically determines the applied removal technique. Before the optimization starts, the facility tests the management interfaces of the original virtual appliance. If the appliance offers them, then the optimization will apply removal during execution, otherwise the system automatically falls back to pre-execution removal.

4.2.3 Validation

The optimization facility requires validation algorithms to ensure that reduced appliances still offer the key functionality of their original appliances. As appliance developers precisely know the key functionality, we assume that they can accompany original virtual appliances with their validation algorithms. For example, to define validation algorithms, developers could reuse unit and integration tests (available because of the software development process [37]). However, these algorithms must evaluate virtual appliances both semantically and functionally. E.g., they could ensure response time requirements or they could also confirm the absence of known security issues. Obviously, validator algorithms cannot be discussed in the article, therefore we assume that these algorithms are available for the appliance developers before they start the optimization process.

Validator algorithms are applied by the $valid : P \times \varphi \rightarrow \{true, false\}$ function. This function calls the appliance's ($p_\sigma \in P$) validator algorithm that evaluates the virtual machine ($vm \in \varphi$) instantiated with the reduced appliance ($vm := initVM(p_{red})$). Where $initVM : P \rightarrow \varphi$ depicts the instantiation of a virtual appliance in a virtual machine. Non-successful validation ($valid(p_\sigma, vm) = false$) leads to the restoration of the original virtual appliance and to the omission of the previously removed item (i_{hw}) from future optimization iterations.

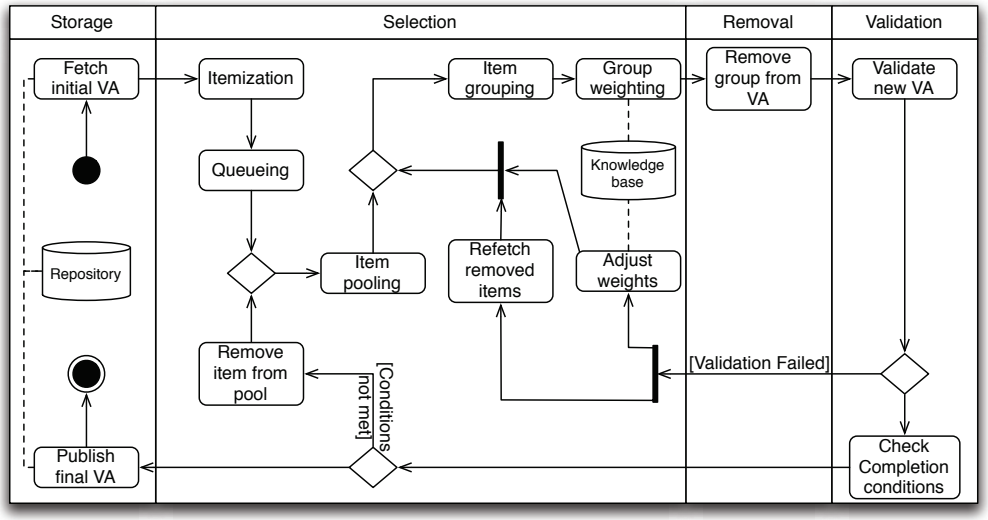


Fig. 3. Overview of the proposed optimization technique

4.2.4 Completion

Before every new optimization iteration, the facility decides if it has reached the completion condition of the appliance developer. The completion condition limits the time and resource usage of the optimization facility and allows the creation of sub-optimally sized appliances.

The decision is made by evaluating the *completion condition* specified as an arbitrary conditional expression based on five metrics: (i) the *number of optimization iterations* executed, (ii) the *current size* of the virtual appliance, (iii) the *size reduction* achieved by the optimization iterations, (iv) the *wall time* for the entire optimization process and (v) the *size of the remaining* (or not validated) items of the virtual appliance.

If the completion conditions have not been met yet, then the optimization facility evaluates the feasibility of creating an *intermediate virtual appliance*: $p_\sigma := p_{red}$. Therefore, the evaluation takes into consideration the cost of the intermediate virtual appliance creation and compares it to the possible gains on future optimization iterations (assuming that the optimization facility will initiate a virtual machine for every non-validated item). As a result, the system reduces the time required for future virtual machine initiations (as the size of p_σ is decreased during the optimization process). In other words, the facility reduces the optimization time by utilizing the effects of optimization through intermediate appliances before reaching the final optimized VA.

The optimization process concludes with the *publication of the final virtual appliance* when the completion conditions are met or there are no more items ($optisize : P \rightarrow \{true, false\}$) to remove from the appliance:

$$optisize(p) := \begin{cases} true & \nexists i \in it(p) : \Big(\big(it(p_x) = it(p) \backslash \{i\}\big) \\ & \wedge \big(valid(p, initVM(p_x)) = true\big)\Big) \\ false & \text{otherwise} \end{cases} \quad (4)$$

During this step the optimization facility first fetches the original virtual appliance from the repository. Then attaches it to the facility's host machine where the successfully validated items are removed from the appliance. Finally, it uploads the locally modified appliance to the repository as an optimized version of the original one.

5 Implementation of the Virtual Appliance Optimization

Fig. 3 depicts the entire optimization process, however it gives details of the selection related operations only, the rest of the operations are discussed in later sections.

5.1 Item Selection

5.1.1 Virtual Appliance Itemization

When the optimization facility receives a request for minimizing a virtual appliance (p_σ), it first *fetches the appliance* from the repository. The disk image of the appliance is analyzed by the *file-based itemization* technique that reads its file system and identifies items (files in the current implementation) of the appliance. During itemization our technique collects and passes the following metadata about each file: (i) the *item size* ($size(i)$), (ii) *dependencies* representing item relations (e.g. inclusion, parent/child relationships) and (iii) creation and modification *timestamps*.

The optimization facility contains an item pool used as a metadata cache to avoid frequent queries on the appliance's file system. This *pool* stores metadata for all items that are ready for the further tasks of the optimization iteration. The *item queue* is used as an intermediary between the file system and the item pool. The queue is the source to fill the unused item capacity of the pool. The length of this queue is automatically determined by the amount of removable items during a single optimization iteration. The maximum value is the same

as the number of virtual machines used for validation as discussed in Section 5.3. If the queue is full, then the itemization procedure is blocked until the optimization facility removes items from the pool. Therefore the item queue remains full while the itemization processes all parts of the appliance.

5.1.2 Grouping

The file-based itemization algorithm produces so many items from a virtual appliance (see Table 3) that creating a virtual machine for each item's removal and validation is inefficient. Therefore, the algorithm groups these items together to decrease the number of removal and validation operations. Grouping has two tasks: (i) form groups from items that are more likely to be removed together and (ii) aggregate the metadata attached to the individual items and present them as group metadata.

We propose three different grouping techniques: (i) directory structure based – items in the same directory will form a common group – , (ii) software package based – items in the same package (e.g. `dpkg`) form a group – and (iii) creation time proximity based – items created within specific time periods belong to the same group.

Item grouping operates on the items in the pool. The algorithm waits until the pool is either full or the itemization has completed. As a result, grouping always handles the maximum amount of items.

If the removal of a group fails the validation then it is split into smaller groups (or items). The previous groupings of an item are stored among its metadata enabling the evaluation of the prior group participation coefficient – see Eq. (7). If the validation fails on an item then the facility saves its metadata as a negative example in the knowledge base. Later, this example helps the calculation of the removal success rate coefficient – see Eq. (8) – during the optimization of other appliances.

Grouping efficiency is measured through the *grouping failure rate* that is the ratio between the number of successfully and unsuccessfully validated groups formed by a specific grouping solution (see Fig. 4). The grouping failure rate can reveal if the applied grouping technique cannot be used for a given appliance. We found that failure rates over 30% indicate the need for switching between grouping techniques. The current implementation uses the directory-based grouping.

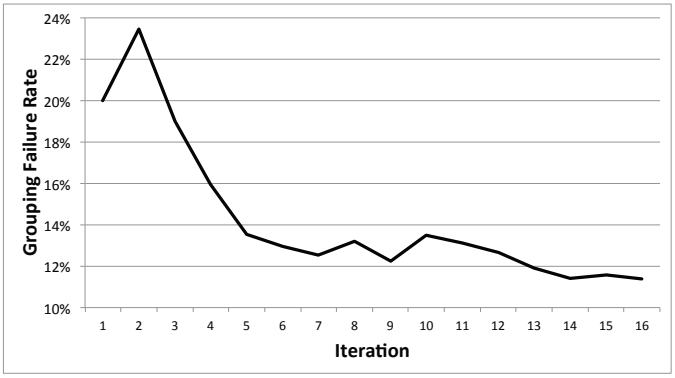


Fig. 4. Typical group failure rate of the directory-based grouping algorithm during optimization

5.1.3 Item Weight Calculation

After grouping, we discuss the *item weighting* step in the selection phase. This step was already introduced in Section 4.2.1. Here only the implemented weight calculation algorithm is described as composite of a base weight function and several coefficients:

$$w_A(i,p) := \gamma(i)\kappa(i)w_S(i,p) \tag{5}$$

Where $w_A : I \times P \rightarrow V$ is the composite weight function of volatile items. This function is based on the size-based weight function ($w_S : I \times P \rightarrow V$). The value of the $w_S(i,p)$ function is modified by the *prior group participation* ($\gamma : I \rightarrow V$) coefficient that prefers items with successfully validated group siblings. The $\kappa : I \rightarrow V$ coefficient favors items with high *removal success rates* of prior optimizations.

First, we define the base weight function that assigns weights considering only the item size ($size(i)$). Consequently, the optimization facility will choose removable items that have higher impact on the appliance size. As a result, the facility significantly reduces the appliance even with highly constrained completion conditions:

$$w_S(i,p) := \frac{size(i)}{\max\limits_{j \in it(p)}(size(j))} \tag{6}$$

Therefore, validation progresses from the largest items towards the smaller ones. As an advantage, this weight value can be calculated without the knowledge base.

Coefficient γ uses information about *prior group participation*. Thus, previously improperly grouped items modify each other's weight. Item metadata stores the previous groupings of an item along with all previous siblings. If an item has participated in a wrong group and some of its group siblings have already passed validation, then their success rate alters the base weight:

$$\gamma(i) := \begin{cases} M(i) > 0 & \min(1, 1 + \frac{M_{success}(i) - M_{faulty}(i)}{M(i)}) \\ M(i) = 0 & 1 \end{cases} \tag{7}$$

Where $M : I \rightarrow \mathbb{N}$ defines the number of already validated siblings in a previously faulty grouping where the current item was a member. $M_{faulty} : I \rightarrow \mathbb{N}$ specifies the number of those siblings that already failed the validation phase. Consequently, $M_{success} : I \rightarrow \mathbb{N}$ is the number of successfully validated siblings.

Coefficient κ alters the base weight value with previous *removal success rate* of the individual items. As a prerequisite, validation results of items from previously optimized virtual appliances are stored in and restored from the knowledge base. With the help of the knowledge base this coefficient encourages the removal of

those items that were previously removed successfully.

$$\kappa(i) := \frac{N_T(i) - N_F(i)}{N_T(i)} \tag{8}$$

Where $N_F : I \to \mathbb{N}$ is the number of unsuccessful removals of a given item, and $N_T : I \to \mathbb{N}$ is the number of trials made on the item.

5.2 Parallel Validation

Fig. 5 exemplifies the items and groups available for validation throughout an optimization process. During the individual iterations the removal action has had more than 400 candidate items (or groups). Our removal and validation technique (see Sections 4.2.2 and 4.2.3) requires an individual virtual machine for the evaluation of each item (or group). However, according to our measurements seen in Fig. 10, virtual machine instantiation is the most time consuming operation of the optimization facility. Therefore, multiple removal and validation tasks has to be executed in parallel. To allow parallelism, the optimization system must be deployed on a cluster capable to execute several virtual machines simultaneously.

Fig. 6 reveals that parallelism in the facility starts after assigning the weight values. First, the system selects the highest weighted groups or items, and for each one of them it initiates a removal and validation task in a dedicated virtual machine (see the "*multiple selection*" action in Fig. 6). As a result, we receive the success reports on several validation tasks in parallel.

These validation success reports are independent from each other. However, the key functionality of the appliance could depend on some interchangeable items (e.g. at least one of the items should be present for the key functionality). To avoid interchangeability problems, the optimization facility creates a group with the union of the successfully removed items. This group is removed from the original appliance and validated (*final validation*) to ensure that removed items does not cause interchangeability problems. On validation success, the group members are removed from the item pool and

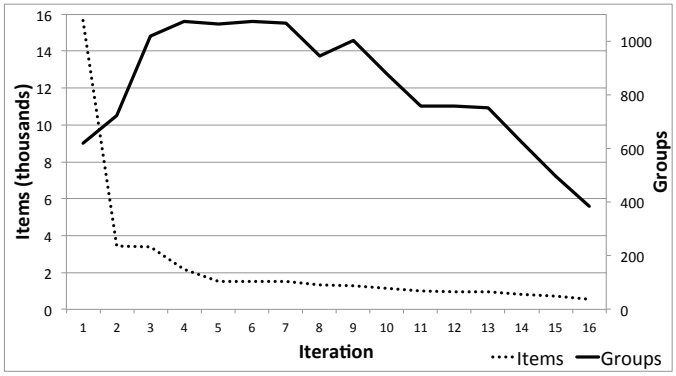


Fig. 5. Number of groups formed from the items available during optimization

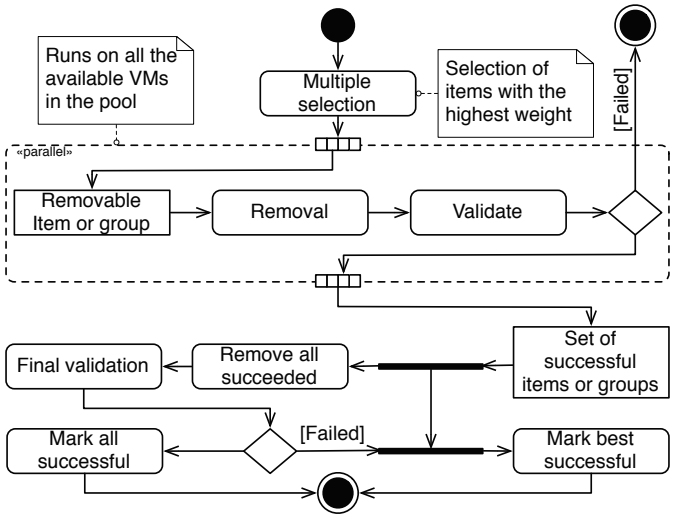


Fig. 6. Parallelism in the validation process

from the appliance. Afterwards, their removal success is added to the positive examples of the knowledge base.

If the final validation fails, the facility selects the highest weighted successful item or group for permanent removal. Other successfully validated items remain in the item pool with a successful validation marker. This enables their early revalidation with the highest weighted element already removed from the appliance.

However, our strategy for selecting permanently removed items may lead to a suboptimal solution. For the optimal selection, the optimization facility should evaluate all possible combinations of the successfully validated removals and mark the combination with the highest cumulative size. The cost of evaluating all possible combinations renders the optimal selection procedure beyond reason and therefore our system never applies it.

5.3 Virtual Machine Management Strategy

Parallel validation requires multiple virtual machines ready to be validated. Therefore, the virtual machine management strategy aims at pre-initializing virtual machines before their actual use by the validation tasks. However, initializing a virtual machine requires substantial amounts of bandwidth and time (see Table 3 and Fig. 10 for details). Therefore, the management strategy defines a virtual machine lifecycle (see Fig. 7) that allows the reuse of previously successfully validated virtual machines for future optimization iterations.

To reduce the delays between removal and validation the algorithm prepares a virtual machine pool that lists those virtual machines that are available for validation. Our strategy automatically determines the size of the pool as the number of available virtualized CPUs in the optimization system. If the optimization system shares the underlying IaaS system with other services then the system administrator of the facility can set up resource usage limitations relative to the entire system.

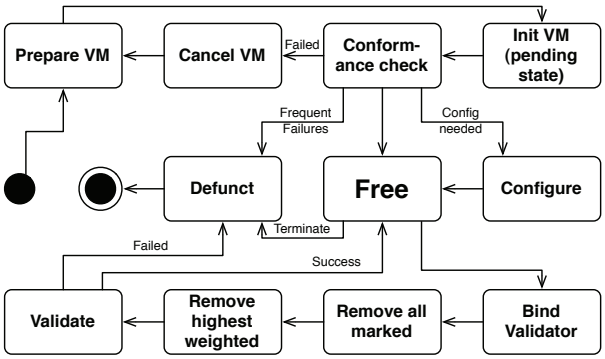


Fig. 7. Virtual machine management states

The first task of the management strategy (designated with the "*Prepare VM*" state) uses the IaaS system to *allocate* as many VMs as possible for the optimization. Each acquired virtual machine is instantiated with the original appliance ($initVM(p_\sigma)$) during its "*Init VM*" state. Then the "*conformance check*" state confirms the accessibility of the created virtual machine to avoid false validation failure reports on improperly initialized virtual machines. If the conformance check fails, then the system repeats the allocation task. Otherwise, the appliances are "*configured*" to run inside their virtual machines. The configuration step handles internal configuration provided by the appliance developer, then it also manages the external network configuration of the virtual machine (e.g. setting up the firewall). After the VMs are initialized they are ready to be used for the removal and validation tasks (detailed in Section 5.2).

When the virtual machine reaches the "*free*" state then it becomes usable for the removal and validation processes. Afterwards, the next state "*binds the validator*" task with a virtual machine. During this phase the virtual machine handler first removes all the previously successfully validated ("*remove all marked*") items from the virtual machine. Then it generates the list of removal requests (e.g. through the management interfaces of the appliance) for the highest weighted removable items (i_{hw}) or groups. Next, the VM enters the "*validation*" state when the handler evaluates the validation algorithms on the virtual machine running the reduced virtual appliance (p_{red}). If both the evaluation and the removal requests were successful then the virtual machine becomes "*free*" – reusable – again.

Faulty validation renders the affected virtual machine "*defunct*". This leads towards the second task of the management strategy: the *recovery* of the defunct VM. This could imply the addition of the previously removed item. However, the addition would require the revalidation of the restored virtual machine. Therefore, instead of trying to recover the defunct VM, the manager terminates it, then initiates a new virtual machine. Next, the manager synchronizes the successfully removed items in the new VM. After recovery, the parallel validation branches compete with each other for the new VMs.

The parallel execution of the validation leads to VMs running slightly more optimized virtual appliances. Therefore, on successful validation, the final task of the manager prepares the content *synchronization* of all the successfully validated VMs to allow their reuse for later validations. The content synchronization is accomplished by generating the list of permanently removable items for the "*remove all marked*" operation.

6 Optimization Results

This section presents the experimental results with the optimization facility: (i) we introduce the methodology and the aim of the experiments; (ii) the testbed infrastructure for the experiments is detailed; (iii) the test virtual appliances are discussed; finally, (iv) we evaluate and analyze the experimental results.

6.1 Methodology

This article represents measurements with the function $\mathcal{M} : \mathcal{F} \rightarrow \mathbb{R}$. This function evaluates its arguments ($\mathcal{F}$ represents an arbitrary function and a specific parameter set) repeatedly and measures the execution time (t_{ev}) for each individual evaluation. Measurements are executed until the sample standard deviation (s_N) of the t_{ev} values becomes stable, thus the value of two subsequent standard deviation calculations are within 1%:

$$\frac{s_N(t_{ev}) - s_{N+1}(t_{ev})}{s_N(t_{ev})} < 0.01 \text{ where } N \geq 2 \qquad (9)$$

Function $\mathcal{M}$ calculates the median of the measured evaluation times of $\mathcal{F}$ after the deviation is stabilized.

6.1.1 Speedup

We have defined the efficiency of the optimization facility with the speedup function – $S : (P) \rightarrow \mathbb{R}$:

$$S(p_\sigma) = \frac{\overbrace{\mathcal{M}(initVM(p_\sigma))}^{\text{baseline}}}{\underbrace{\mathcal{M}(initVM(p'_\sigma))}_{\text{optimized}}} \qquad (10)$$

Speedup ($S(p_\sigma)$) is the ratio of the measured instantiation times of a service package before (p_σ) and after (p'_σ)

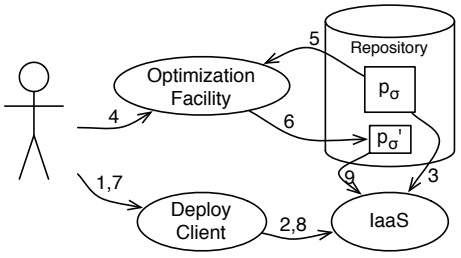


Fig. 8. Efficiency of size optimization

the size optimization was applied. We consider the optimization algorithm successful if the optimized virtual appliance can be instantiated faster than the original:

$$S(p_\sigma) > 1 \tag{11}$$

The *baseline* measurement ($\mathcal{M}(initVM(p_\sigma))$) for the speedup values is presented in *step 1-3* in Fig. 8. During this measurement, we used the deployment client to request the IaaS system for the instantiation of p_σ.

Fig. 8 reveals that the baseline measurement is followed by the request to optimize the original appliance (p_σ – *steps 4-5*). After the completion of the optimization process, the final reduced appliance (p'_σ) is stored in the repository (*step 6*). Finally, the testbed is ready to evaluate the speedup by measuring the instantiation time of p'_σ in *steps 7-9* – $\mathcal{M}(initVM(p'_\sigma))$.

6.1.2 Cost Efficiency

There is a tradeoff applying the technique introduced in this article. For example, before any user initiated deployment could occur, the optimization process creates virtual machines on purpose to allow the validation of the reduced appliance. To evaluate this tradeoff, we measure the time spent during the optimization process according to *steps 4-6* in Fig. 8. Then, we calculate the number of virtual machine instantiations (or user initiated deployments – N_{dep}) required to compensate the time spent on the appliance size optimization task:

$$N_{dep}(p_\sigma) := \frac{\mathcal{M}(optimize(p_\sigma, \emptyset))}{\mathcal{M}(initVM(p_\sigma)) - \mathcal{M}(initVM(p'_\sigma))} \tag{12}$$

By default, the optimization – and therefore the measurement – is executed until $p'_\sigma := optimize(p_\sigma, \emptyset)$ reaches its minimal size: $optisize(p'_\sigma) = true$.

6.2 Testbed Infrastructure

The optimization facility is designed for implementation on top of IaaS systems (e.g. Eucalyptus [21] or Virtual Workspaces Service [38] that is part of Nimbus). In this article, we discuss an implementation based on Eucalyptus, because of its two advantages: (i) Eucalyptus supports all functionalities of the optimization facility without extensions or modifications, and (ii) Eucalyptus could be easily replaced with Amazon EC2 [20] to present the viability of the optimization facility on a commercial IaaS cloud system.

Fig. 9 shows the infrastructure used for the experiments. In this infrastructure there is a single Internet connection available only for the Eucalyptus *HeadNode*. This node runs as the controller of the local cluster by managing both its IaaS behavior and also its networking – e.g. DHCP, DNS and routing. We also have the internal Eucalyptus repository, Walrus, installed on this node. Finally, this node also hosts the optimization facility itself in order to have a low latency and high bandwidth connection with the IaaS system and its repository. This

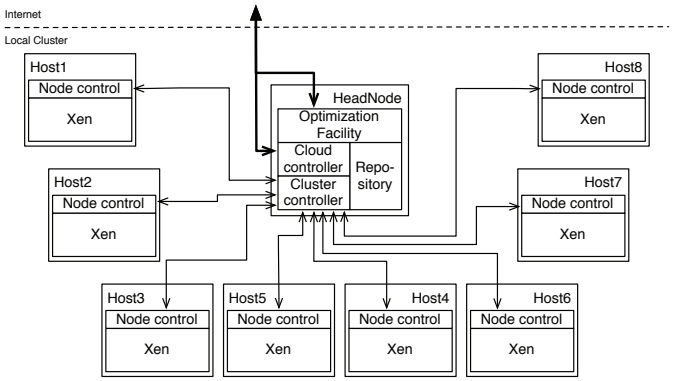


Fig. 9. The testbed

infrastructure satisfies the facility's requirement to have a high bandwidth connection with the repository.

The remaining hosts are configured with the same software. They run the Xen virtual machine monitor [6] in order to enable virtual machine creation and management on the nodes. They also execute the Eucalyptus node controller that enables the remote access to the locally available virtual machine monitor. All the nodes have the same hardware configuration (4 CPUs, 4GB of RAM and 80GB of HDD) and they are connected with gigabit Ethernet towards the HeadNode.

6.3 The Experimental Virtual Appliances

In this subsection, we outline the virtual appliances selected to test the optimization facility. Based on the popularity of basic Internet services we have selected and defined two appliances: the SSH and the Apache web server appliance. The key functionality of the *SSH* appliance (see Table 1) enables a user to transfer shell scripts to the machine and allows their execution. In contrast, the key functionality of the *Apache* appliance (see Table 2) does not allow arbitrary code execution, however it enables users to upload static html content that

TABLE 1
The definition of the SSH virtual appliance

Key functionality	The SSH virtual appliance should offer a virtual machine image that provides *remote execution and transfer* capabilities. Therefore, this is a general-purpose virtual appliance that enables the execution of arbitrary code. As a prerequisite, the user has to transfer the executable and its dependencies before execution.
Construction	1) Create a virtual machine image for Xen with xen-tools of debian. 2) Start up the created VM image. 3) Add the ssh daemon and the rsync transfer utility. 4) Enable remote ssh based root logins in the VM.
Validator algorithm	1) Create a shell script that prints out "hello world". 2) Transfer the previously created shell script to the target virtual machine with rsync. 3) Remotely execute the transferred script. 4) Check whether the execution returns with "hello world". If not then the virtual machine is not valid.

TABLE 2
The definition of the Apache virtual appliance

Key functionality	The aim of the Apache appliance is to provide an HTTP server that is capable of *serving static html pages* to its users. As a prerequisite, the user has to transfer the static html pages that should be offered by the server.
Construction	1) Create a virtual machine image for Xen with xen-tools of debian. 2) Start up the created VM image. 3) Add the ssh, rsync and apache daemons.
Validator algorithm	1) Create an HTML document that has an html header with the text of "hello world". 2) Transfer the previously created HTML document to the apache web server's folder on the target VM. 3) Download the pre-transferred HTML file with an HTTP request. 4) Check for the html header with the text "hello world". If the header is not present then the target virtual machine is not valid.

can be served later by its hosting machine. Tables 1 and 2 also describe the validation algorithms for their corresponding appliances; these algorithms were simplified for demonstration purposes. The proposed system does not limit developers in algorithm design so they can include more functional requirements and even non-functional ones like security related checks.

Both virtual appliances were based on xen virtual machine images created with the `xen-create-image` tool offered by Debian Linux. We have altered the images by installing the necessary Debian packages to support SSH or Apache. We have configured both virtual appliances with 100 MB of free space to allow custom content for their users. The free space is big enough to allow the initial use of the appliance, however for advanced usage it either has to be extended with one of the available tools (e.g. `resize2fs`, `xfs_growfs`) before publishing the appliance or alternatively a new file system can be attached for the custom content (e.g. by attaching an elastic block store – EBS – volume on Amazon EC2). As several commercial cloud providers require the use of new file systems for persistence, developers would similarly configure their appliances as we did with the test appliances. Neither of these virtual appliances require configuration during their instantiation, because they are standalone services and they do not depend on external network connections. The properties of the original virtual appliances are listed in Table 3.

This article does not discuss optimization results with other virtual appliances because of the generic and widespread use of the previously introduced ones. However, we have already discussed the application of the optimization facility on the biochemical application of TINKER in our previous paper [39].

6.4 Evaluation

6.4.1 Basic Optimization Results

First, we have executed the optimization process on the previously selected virtual appliances without any completion condition (see Section 4.2.4 for details). Therefore, the optimization facility executed the optimization task until it exhausted the removable items from the virtual appliances. The resulting appliances are presented in Table 4. The eight machines optimized the appliances to less than tenth of their original size within no more than two hours. In addition, appliance file counts have dropped to less than $\frac{1}{50}th$ of their originals. In case of the test appliances, the removed files included documentation, unused libraries and executables.

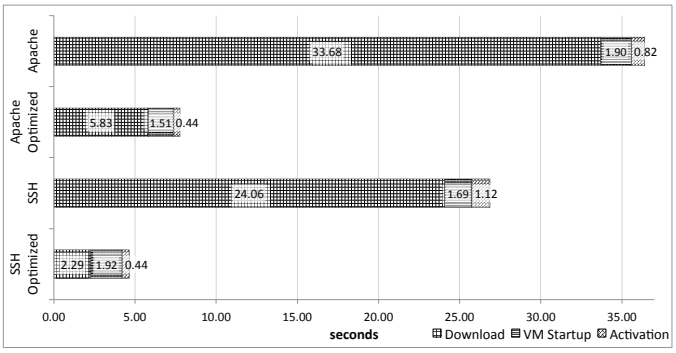


Fig. 10. Instantiation time before and after optimization

TABLE 3
Basic properties of the test appliances

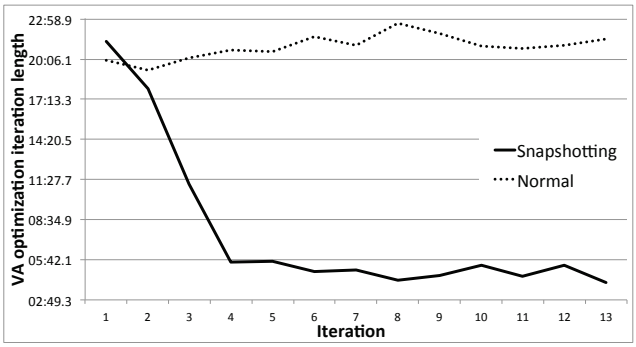
Appliance	Compressed Size	Number of Files
SSH	120MB	10647
Apache	165MB	14050

To allow comparison, Table 4 also lists the properties of the rPath [13] Apache appliance [40] (a typical appliance created with the pre-optimizing algorithm of rPath – see Section 2). This appliance is a typical developer created appliance that does not take into consideration the key functionality of the appliance therefore even after optimization it contains non-necessary content.

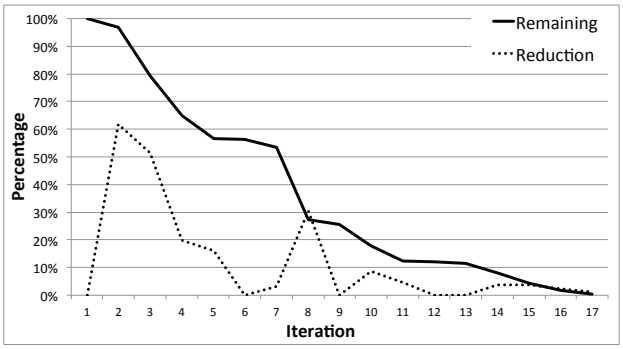
Then, we deployed both the optimized and the original virtual appliances and measured the execution time of the different instantiation steps. These results are presented in Fig. 10 according to the three steps of virtual appliance instantiation introduced in Section 4. Table 4 proves that we reached significant speedup on the instantiation time of the original virtual appliances. After the comparison of the service activation times of the original and the optimized appliances, we conclude that the optimization facility successfully removed items causing unnecessary delays in activation (e.g. some unused system level services that were not important for the key functionality of the appliance). This feature increases the security of the optimized virtual appliances compared to the original ones, because the size-optimized appliances cannot be attacked through unused system level services.

6.4.2 Increasing Optimization Efficiency

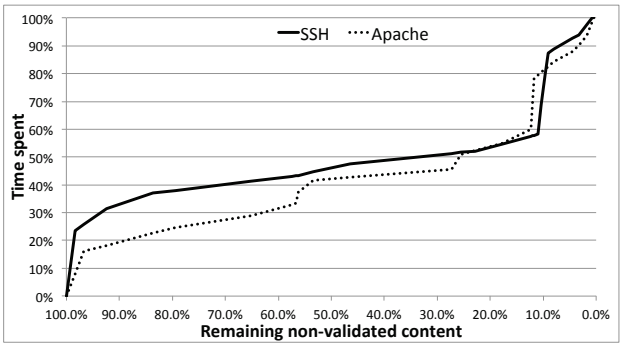
For the purpose of the evaluation, special optimization processes were initiated that monitored the state of



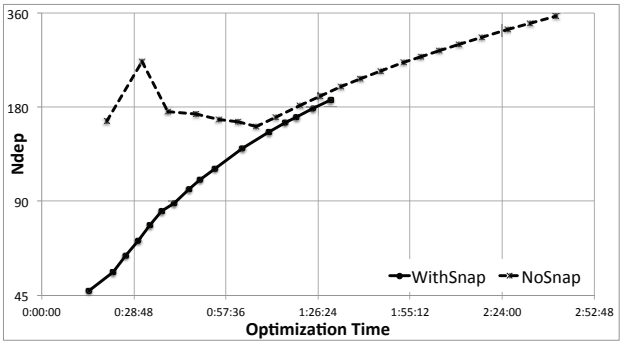
(a) Effects of intermediate virtual appliance creation



(b) Stability of completion conditions



(c) Effects of *remaining* size completion condition on execution time



(d) Required number of future instantiations to overcome the optimization time of the Apache appliance

Fig. 11. Increasing the effectiveness of the optimization facility

TABLE 4
Basic virtual appliance properties of the optimized test appliances

Appliance	Compressed Size	Files	Opt. time	$S(p_\sigma)$
SSH′	6.6MB	197	4958 secs	4.72×
Apache′	13MB	236	7468 secs	5.85×
rPath Apache	152MB	28923	N/A	N/A

the optimization system at the end of the optimization iterations (see Section 5.1). The gathered data includes (i) the items removed, (ii) the current value of the five completion condition metrics (defined in Section 4.2.4), (iii) the number of virtual machines used, (iv) the number of validations passed, (v) the average time to initiate a virtual machine, etc. Based on these values, the system's behavior can be evaluated without executing the optimization process in too many configurations.

As it was depicted in Table 4, the unlimited optimization process requires hours to complete. We have implemented a snapshotting technique that creates an *intermediate virtual appliance* after the optimization iterations as discussed in Section 4.2.4. Fig. 11a presents the optimization of the SSH virtual appliance with and without the creation of the intermediate virtual appliances. The figure reveals that intermediate VAs can dramatically decrease (e.g. from twenty minutes to less than five) the time and cost of later optimization iterations. As a result, this technique immediately results in the reduction of the total optimization time.

However, this approach still does not exploit the completion condition evaluation. We have observed, that the optimization process could not reach significant size reduction in its late stages (the size of the remaining non-validated items follows a Pareto distribution). Thus, to avoid the tail problem, we have investigated the various completion conditions (introduced in Section 4.2.4) whether they can predict the inefficiencies of the last phase of the optimization process. Unfortunately, most of the completion conditions cannot predict inefficiencies, because they are either not stable enough throughout the entire optimization process (see *reduction* in Fig. 11b), or their inefficiency threshold cannot be generalized for multiple appliances (e.g. optimization time). A stable completion condition has to be monotonic, in order to allow the definition of a threshold value that the completion condition variable only crosses once during the optimization process. Consequently, we define *inefficiency indicators* as special completion condition variables that reveal the inefficiency of the optimization process after passing a predefined threshold value.

From the list of Section 4.2.4, we have identified two completion condition variables as candidates for the role of inefficiency indicator. The first one is the *size reduction* percentage achieved during a single iteration.

The second one is the percentage of the cumulative size of the *remaining* (non-validated) items compared to the current size of the intermediate virtual appliance. Fig. 11b presents the changes of these two completion conditions throughout an entire optimization process. We have chosen the *remaining* completion condition for further investigation because its monotonic nature.

Fig. 11c demonstrates the effects of applying different *remaining* completion condition variables. In this figure, we have normalized the optimization time for better comparison (for the explicit optimization time values see Table 4). In order to decrease optimization time but still maintain adequate appliance size we have identified that the *remaining* completion condition variable can be used as an inefficiency indicator with the threshold *10%*. This threshold reduces the optimization time by 40% and still maintains close to optimal virtual appliance sizes.

Finally, using the previously introduced options for increasing effectiveness we have calculated the N_{dep} values (see Eq. 12) based on the statistical information collected during the execution of the optimization processes. Fig. 11d presents the calculated values and presents the minimum amount of the future instantiations required before the optimization becomes profitable. According to the figure, the cost of the optimization procedure is high in the early stages reflecting the high grouping failure rates (see Fig. 4) and the long iteration lengths (see Fig. 11a). Later, the cost increases linearly with the executed optimization iterations because the last iterations of the process are not reducing the instantiation time considerably.

7 Conclusions

In this article, we have outlined the challenges of using virtual appliance based deployment in highly dynamic service environments and Infrastructure as a Service cloud systems. We have shown how to address these challenges by virtual appliance size optimization. The proposed approach uses active fault injection to remove parts of virtual appliances. The reduced virtual appliances are validated with appliance developer provided validator algorithms in order to maintain the key functionality of the appliance. The algorithm also includes several item selection and grouping techniques in order to decrease the number of validation steps required to achieve the optimized virtual appliance.

The proposed size optimization approach offers several advantages over the existing solutions. First, the appliance developer does not need to know the dependencies of the service that it plans to encapsulate in a virtual appliance, instead this article assumes the appliance developers can define the key functionality of their desired appliance in the form of validation algorithms. Second, this solution could also be used with existing virtual appliances. Thus, it can significantly reduce the operating costs of those appliances that are already used in highly dynamic service environments. The proposed technique not only reduces the virtual machine instantiation time, but it also provides a technique that minimizes the virtual appliance optimization time and allows the early release of the optimal appliances.

We have presented our implementation by experimenting on two well-known services encapsulated in virtual appliances. The results revealed that the size of typical virtual appliances could be significantly optimized. Based on these experiments, we have also identified that the time and cost efficiency of the optimization algorithm can be improved by (i) creating intermediate virtual appliances and by (ii) terminating the optimization process using the ratio of the remaining non-validated items in the suboptimal VA as the completion condition.

Future research considers the further optimization of the item selection and grouping techniques. We are also considering research on more efficient scheduling algorithms for the parallel validation phase in order to reduce the resource usage caused by the revalidation of previously successful validation results. We also plan to investigate the security related effects of the proposed optimization approach including the effects of virtual machine reuse for multiple removal and validation phases and whether the optimized appliance is more vulnerable than the original.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," University of California at Berkley, Tech. Rep. UCB/EECS-2009-28, Febr. 2009.

[2] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, June 2009.

[3] N. Susanta and C. Tzi-Cker, "A survey on virtualization technologies," ECSL-TR-179, Stony Brook University, Tech. Rep., Febr. 2005.

[4] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[5] Vmware. [Online]. Available: http://www.vmware.com

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebar, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[7] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum, "Virtual appliances for deploying and maintaining software," in *LISA '03: Proceedings of the 17th USENIX conference on System administration*. Berkeley, CA, USA: USENIX Association, 2003, pp. 181–194.

[8] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *SIGCOMM Computer Communication Review*, vol. 39, pp. 50–55, Dec. 2008.

[9] L. Youseff, M. Butrico, and D. Da Silva, "Toward a unified ontology of cloud computing," in *Grid Computing Environments Workshop, 2008. GCE'08*. DOI: 10.1109/GCE.2008.4738443: IEEE, 2009, pp. 1–10.

[10] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environments Workshop, 2008. GCE'08*. DOI: 10.1109/GCE.2008.4738445: IEEE, 2009, pp. 1–10.

[11] B. Benatallah, Q. Sheng, and M. Dumas, "The self-serv environment for web services composition," *IEEE Internet Computing*, vol. 7, no. 1, pp. 40–48, 2003.

[12] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl, "A journey to highly dynamic, self-adaptive service-based applications," *Automated Software Engineering*, vol. 15, no. 3, pp. 313–341, 2008.
[13] rPath - rBuilder, http://www.rpath.com/rbuilder/.
[14] M. Sußkraut, S. Creutz, and C. Fetzer, "Fast fault injections with virtual machines," in *the Fast Abstracts track of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Edinburgh, UK, June 2007.
[15] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Trans. Softw. Eng.*, vol. 16, no. 2, pp. 166–182, 1990.
[16] J. A. Clark and D. K. Pradhan, "Fault injection," *Computer*, vol. 28, no. 6, pp. 47–56, 1995.
[17] S. Bruning, S. Weißleder, and M. Malek, "A fault taxonomy for service-oriented architecture," in *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium*, 2007.
[18] G. Kecskemeti, P. Kacsuk, G. Terstyanszky, T. Kiss, and T. Delaitre, "Automatic service deployment using virtualisation," in *Proceedings of 16th Euromicro International Conference on Parallel, Distributed and network-based Processing*. IEEE Computer Society, Febr. 2008.
[19] G. Kecskemeti, G. Terstyanszky, P. Kacsuk, and Z. Nemeth, "An approach for virtual appliance distribution for service deployment," *Future Generation Computer Systems*, vol. 27, no. 3, pp. 280–289, Mar. 2011.
[20] Amazon Web Services LLC. (2009) Amazon elastic compute cloud. [Online]. Available: http://aws.amazon.com/ec2/
[21] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system." in *CCGRID*, F. Cappello, C.-L. Wang, and R. Buyya, Eds. IEEE Computer Society, 2009, pp. 124–131.
[22] T. Zhanga, Z. Dua, Y. Chenb, X. Jic, and X. Wang, "Typical virtual appliances: An optimized mechanism for virtual appliances provisioning and management," *The Journal of Systems and Software*, vol. 84, pp. 377–387, 2011.
[23] K. Wang, J. Rao, and C.-Z. Xu, "Rethink the virtual machine template," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Newport Beach, California, USA, Mar. 2011.
[24] D. Geer, "The os faces a brave new world," *Computer*, vol. 42, pp. 15–17, Oct. 2009.
[25] Public Amazon Machine Images, http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=171, 2010.
[26] VMWare public virtual appliances, "http://www.vmware.com/appliances/," 2010.
[27] SUSE Appliance tookit, "Suse galery," http://susegallery.com/, Jan. 2011.
[28] Science Clouds, "http://scienceclouds.org/marketplace/," Jan. 2011.
[29] H. A. Basit and S. Jarzabek, "Detecting higher-level similarity patterns in programs," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, Lisbon, Portugal, Sept 2005, pp. 156 – 165.
[30] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," *SIGPLAN Notices*, vol. 40, no. 10, pp. 167–176, Oct. 2005.
[31] Vizioncore Inc., "voptimizer, optimization of virtual machine size and performance," 2008.
[32] G. Kecskemeti, G. Terstyanszky, P. Kacsuk, and Z. Nemeth, "Improving virtual appliance deployment using minimal manageable virtual appliances," ***Submitted:*** *IEEE Transactions on Computers*.
[33] A. Kertész, G. Kecskeméti, and I. Brandic, "An sla-based resource virtualization approach for on-demand service provision," in *Proceedings of the International Conference on Autonomic Computing, 3rd International Workshop on Virtualization Technologies in Distributed Computing*, 2009, pp. 27–34.
[34] D. Blackman, "Debian package management, part 1: A user's guide," *Linux Journal*, vol. http://www.linuxjournal.com/article/4352?page=0,0, Dec. 2000.
[35] W. Vambenepe, "Services distributed management: Management using web services (muws 1.0)," web, Aug. 2005. [Online]. Available: http://docs.oasis-open.org/wsdm/wsdm-mows-1.1-spec-os-01.pdf
[36] S.-M. Yoo, J. W.-K. Hong, J.-G. Park, C.-W. Ahn, and S.-W. Kim, "Performance evaluation of wbem implementations," *KNOM Review*, vol. 8, no. 2, p. 7, Febr. 2006.
[37] A. Bertolino, "Software testing research: Achievements, challenges, dreams," *Future of Software Engineering*, pp. 85–103, 2007.
[38] K. Keahey, I. Foster, T. Freeman, X. Zhang, and D. Galron, "Virtual workspaces in the grid," ANL/MCS-P1231-0205, 2005.
[39] A. Kertesz, G. Kecskemeti, and I. Brandic, "Autonomic sla-aware service virtualization for distributed systems," in *Proceedings of the 19th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2011)*. Ayia Napa, Cyprus: IEEE, Febr. 2011.
[40] rPath, "Apache appliance," http://www.rpath.org/project/aa/releases, 07 2010.

Gabor Kecskemeti has received his MSc degree from the Institute of Information Technology at the University of Miskolc, in 2004. He is currently working on his Ph.D. in the Centre of Parallel Computing at the University of Westminster. He also participates in the research of the Laboratory of Parallel and Distributed Systems at MTA-SZTAKI, Hungary. He has got involved in several successful grid related projects and research resulting the P-Grade Grid Portal of MTA-SZTAKI and the Grid Execution Management for Legacy Code Applications (GEMLCA) of the University of Westminster.

Gabor Terstyanszky is Reader in Distributed Systems at the University of Westminster, London, United Kingdom. He is the co-leader of the Centre for Parallel Computing at the University of Westminster. He received his MSc degree at the Electrotechnical University, St. Petersburg, Russia. He obtained his MPhil and Ph.D. degree at the University of Miskolc, Hungary in 1990 and 1997, respectively. His research interests cover distributed and parallel computing systems including clouds, desktop and service grids. He was involved in more than 20 research projects either as Local Coordinator or Principal Investigator. He has published more than 100 papers in periodicals and conference proceedings on distributed and parallel computing systems. He was member of Program Committee of several European and world conferences. He was also Guest Editor of several special issues published in periodicals on distributed computing infrastructures.

Peter Kacsuk is the Head of the Laboratory of Parallel and Distributed Systems in MTA SZTAKI Computer and Automation Research Institute of the Hungarian Academy of Sciences. He received his MSc and university doctorate degrees from the Technical University of Budapest in 1976 and 1984, respectively. He received the kandidat degree (equivalent to Ph.D.) from the Hungarian Academy in 1989. He habilitated at the University of Vienna in 1997. He received his professor title from the Hungarian President in 1999 and the Doctor of Academy degree (DSc) from the Hungarian Academy of Sciences in 2001. He has been a part-time full professor at the Cavendish School of Computer Science of the University of Westminster and the Eotvos Lorand University of Science Budapest since 2001. He has published two books, two lecture notes and more than 200 scientific papers on parallel computer architectures, parallel software engineering and Grid computing. He is co-editor-in-chief of the Journal of Grid Computing published by Springer.