2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Scalable System Environment Caching and Sharing for Distributed Virtual Machines

Teng Long*, Ilchul Yoon†, Alan Sussman*, Adam Porter* and Atif Memon*
*Department of Computer Science, University of Maryland, College Park, USA
{tlong,als,aporter,atif}@cs.umd.edu
†Department of Computer Science, State University of New York, Incheon, South Korea
icyoon@sunykorea.ac.kr

***Abstract*—Virtual machines have become very widely used in many software development communities. Developers can conveniently provision specific machine configurations using VMs, and those VMs can contain operating systems, libraries, and other applications required to build and execute their software under development. However, the size of full VM images and network bandwidth limitations makes large-scale reuse of VMs among distributed groups of developers extremely difficult. In this paper we address the problem of provisioning software configurations realized as virtual machine images rapidly and incrementally from a set of pristine VM states, by caching and sharing configuration fragments between developer groups. We first formally model the entire configuration space that might be cached and shared between groups, describe the design of our infrastructure to incrementally provision configurations from its fragments, and finally evaluate the performance of our approach with an example scenario common in software testing. Our experimental results show that our approach can help developers reduce the time and resource requirements for provisioning software configurations.**

I. Introduction

Virtual machine images have recently been used for many purposes, such as preserving and distributing operating system states [1], encapsulating and exchanging software system configurations [2], and being provided to end users in an IaaS infrastructure as the initial state of a cloud environment [3]. In many instances the content of a base system in a virtual machine is updated incrementally, and being able to cache and share the intermediate states created by each update is essential for some applications and tools that rely on virtual machines [2], [4].

Currently, such intermediate system states are usually captured and shared as full virtual machine images. The size of such files is likely to be very large (i.e. GBytes), which makes it infeasible to cache a large number of them in a local repository, or to move them frequently across a wide-area network. For example, for automated software testing tools that rely on virtual machine caching and reusing to test software functionality and interoperability over multiple software configurations, the number of states being cached can greatly affect performance [5]. Performance improvements can be large especially when there are dependencies between software developed by independent groups of developers. In that case, the developers can greatly decrease testing time and effort by sharing and reusing many configurations that are identical and required for testing their software [6]. So collaborative testing of software by geographically distributed developer groups is a major usage case for the methods described in this paper. For collaborative software testing it is critical to employ a technique that reduces the space required for saving software configurations realized in cached virtual machine images.

We have observed that in many usage cases of sharing and caching virtual machine images, the content encapsulated in a system image can be divided into (1) a pristine state of the system (often a base operating system installation) and (2) an incremental update from that pristine state. All cached virtual machine images originate from a small number of pristine system states. The pristine systems are usually very large, while the incremental updates to them are relatively small. With proper tools and infrastructure, it is possible to cache and share only the incremental parts to the pristine states. This enables storing many system states in a fixed amount of space, and rapidly reconstructing the states by retrieving only the incremental updates from a local repository or across wide-area network, when a tool or application wants to share the states.

In this paper, we formally model the procedure of incremental updates to pristine system environments. We have also developed an infrastructure called *Ede* (**E**nvironment **D**ifferencing **E**ngine (pronounced the same as Eddy) to support efficient system environment differencing and scalable system environment sharing between distributed sites. *Ede* contains a set of tools to acquire the incremental parts of a system from a pristine state, and to restore the system state from a pristine state using an incremental update. *Ede* also operates a centralized repository for clients in distributed sites to share and request desired VM environments efficiently via the Internet. We apply this infrastructure to a software testing scenario and evaluate the benefits obtained by caching and sharing incremental parts of system environments instead of full virtual machine images. Experimental

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.98

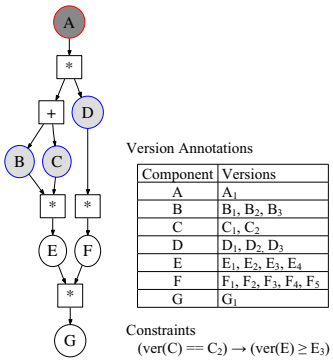

Figure 1. An Example System Model

data suggests that developers can save significant time and space using this infrastructure.

The rest of the paper is organized as follows. Section II is an overview of component-based software testing that in particular relies on environment (virtual machine) caching and sharing for good performance. Our techniques that reduce the size of cached virtual machine states have been developed mainly to support this type of software testing, although the techniques are applicable for other purposes, as we have described earlier. Section III formally models the process of applying incremental updates to pristine system environments in virtual machines. Section IV introduces the infrastructure we designed to support scalable environment caching and sharing. In Section V we describe how the infrastructure can be used to support the software testing scenario from Section II and evaluate the performance benefits from using the infrastructure. Section VI discusses related research, and we wrap up in Section VII with conclusions and a discussion of future work.

II. Background and Motivation

In this section we introduce the process of collaborative building and functional testing of component-based software systems, which is the scenario of using virtual machine images to cache and share system environments for the testing. Section II-A addresses our prior work on modeling component-based software systems, Section II-B introduces *Rachet* [2], an automatic build test system that relies on virtual machines to test component-based systems in a parallel and distributed manner, and finally Section II-C describes a scenario of reusing prebuilt virtual machine images to conduct collaborative functional testing.

A. Annotated Component Dependency Model

We model component-based systems using a representation called ACDM (Annotated Component Dependency Model) [2] [5]. The model consists of two parts: a directed acyclic graph called *Component Dependency Graph(CDG)* and a set of *Annotations*. As illustrated in Figure 1, circular nodes in a CDG represent uniquely identifiable components,

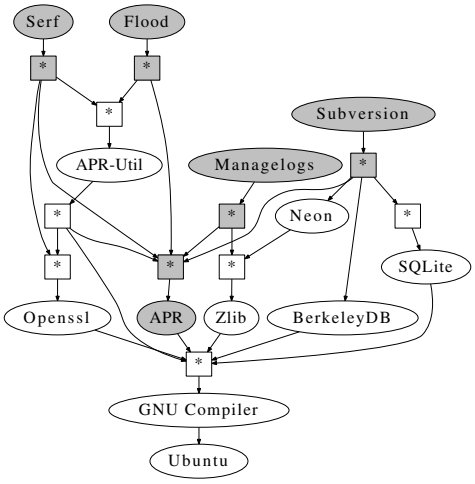

Figure 2. Systems with Common Components

and inter-component dependencies are modeled by connecting nodes with **AND**(*) or **XOR**(+) relationships represented as rectangular nodes. For example, in Figure 1 component `A` *depends on* component `D` and either one of `B` or `C`. Dependency between components means that a component requires other components at build-time, run-time, or both. *Annotations* in this example include version identifiers for components, and constraints between different components and component versions, written in first-order logic.

When different software systems share components, relationships between the systems are represented by an integrated CDG with overlapping sub-graphs. The example CDG in Figure 2 shows that four top-level components (*Serf, Flood, Subversion* and *Managelogs*) depend on different sets of components that *provide* features required by the components. However, there are overlaps between the set of *provider* components. Note that the *APR* component is required by all top-level *user* components. This suggests that all top-level component developers will use their test resources to build the components contained in the shared sub-graph, rooted at the *APR* node, and moreover they will test the behavior of the overlapped components to ensure error-free builds of the top-level component. We will discuss more details about the testing procedure for this scenario in Sections II-B and II-C.

B. Testing Component-based Systems in Parallel

As presented in Figure 1, individual components can have many versions and this will result in a large number of component version combinations on which the correctness of the build and functionality of the top-level component have to be tested. Each combination is called a *configuration*. In our previous work, we developed *Rachet*, an automatic tool to perform the build testing of component-based software systems. Rachet utilizes virtual machines hosted in multiple compute nodes of a high-performance compute cluster or a private cloud to build and test configurations.

Building each configuration individually can be very time-consuming and is unnecessary. First, there could be identical components to build for two different configurations. For example, in Figure 1, building configuration $\{G_1E_1F_1B_1C_1D_1A_1\}$ and $\{G_1E_1F_1B_1C_1D_2A_1\}$ involves building the same set of components $\{G_1E_1F_1B_1C_1\}$ at the beginning of their build paths. *Rachet* therefore uses only one VM to build these components, then make a copy of the virtual machine image and continues with the two branched building proceduress in two VMs. On the other hand, if there exists a virtual machine image in the cache that contains the set of components $\{G_1E_1F_1B_1C_1\}$, *Rachet* will reuse that VM image and only build the missing set of components in that VM. In [5], we showed that the number of virtual machine images that can be cached significantly reduces the overall testing time.

However, the cache available locally on each compute node is limited and therefore it is infeasible to cache a large number of virtual machine images locally, nor can we transfer such images from caches of other *Rachet* processes. It takes minutes to transfer a virtual machine image even on a LAN environment. This means that building some components have to be repeated multiple times, because Rachet will rebuild components if the cost of reusing cached configurations is higher. If we have a cost-effective method to reduce the size of data required to reconstruct each cached virtual machine image, significant time would be saved by avoiding redundant component builds.

C. Collaborative Testing across Distributed Sites

Rachet tests the build-compatibility of a component-based system over many configurations in parallel, at a local cluster environment. However, since components required for such a system are in many cases developed by different parties, testers of these components may also run their own tests at their local sites. Our previous research [6] showed that redundant test efforts were spent by software testers. Similar to reusing locally cached virtual machine images, testers would be able to save build effort if they can share their cached build artifacts efficiently via the network.

There are two challenges preventing testers from sharing build artifacts. First, sharing whole virtual machine images is infeasible because of the bandwidth limit of wide-area network. Second, different testers may use different virtualization techniques, thus shared virtual machine images may not always be compatible with local virtual machine hosting systems. To enable collaborative testing across distributed sites, we need a method that greatly reduces the size of the test environment data being shared, and also allows the reuse format independent from specific virtualization systems used.

III. Problem Modeling

We now model incremental updates to system environments in virtual machines.

Definition *Update*: an update to a system state is defined as an *atomic* and *deterministic* operation that causes a change to the state of the system.

Definition *Pristine State*: the pristine state of a system is an arbitrary state that a system maintainer chooses as the base state before any updates. Any update to the pristine state is considered an incremental update to the pristine state.

We model the states of system environments as a deterministic state machine. The same system must always start from the same pristine state, so that it can be updated deterministically by applying a set of updates. Each update will change the state of the system environment to another known state. Therefore when different users apply the same set of updates to the same pristine system environment, they will create identical states of the environment. Therefore we make the following claim:

A state of a system environment can be deterministically reconstructed by applying a series of updates onto a *pristine* state.

A user **A** who shares the same set of system environment pristine states and the same update space with another user **B** does not need to send the whole system environment to **B** in order to reconstruct a given environment state as long as the user **B** already has the identical pristine state. Instead **A** just needs to send the set of updates that should be applied to the pristine state locally available in *B*. **B** can then *replay* the updates onto its pristine state to reconstruct the desired environment state.

The cost of producing updates for the first time will be much more expensive than replaying the updates later. The tester who generates an update previously non-existent will have to first build a component from its source code and then deploy produced files including the binary code. However, for replaying the update, testers will only need to acquire the update and deploy the files included in the update into their system environments. This update sharing and incremental replaying at distributed sites will greatly reduce overall test efforts.

IV. Infrastructure Design

In this section we describe an infrastructure called *Ede* (Environment Differencing Engine) and apply the incremental update approach to the component-based software testing scenario discussed in Section II.

As defined in Section III, the *update* operation is embodied as building and deploying a component on a system environment. We start by explaining methods that can identify the parts of a *pristine state* realized as a virtual machine image, extract incremental parts from the pristine

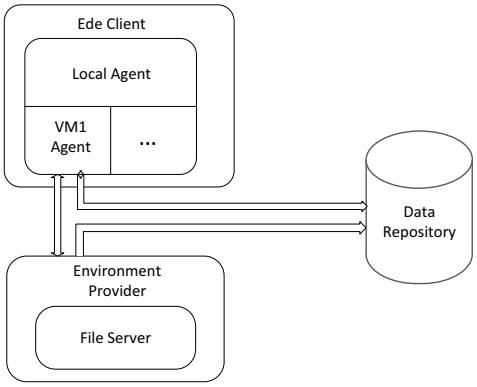

Figure 3. High-level Design of Ede Infrastructure

state once the system is modified by building and deploying components, and finally apply the updates to another pristine state. We also describe the design of an online repository that enables users to efficiently and effectively share the data on reusable system environments. The repository can answer the queries on system environment states that contain specific sets of deployed components. The *Rachet* build testing tool is updated to support the caching and sharing of artifacts that contain incremental state data instead of full virtual machine images.

The high-level design of *Ede* is shown in Figure 3. *Ede* can have multiple clients and environment providers, and the system in Figure 3 has one of each. Every *Ede* client has two parts: a *local agent* and *virtual machine(VM) instance(s)*. The local agent acquires pristine virtual machine instances, controls the state of each VM instance managed by the agent by applying the *update* operations. The local agent accomplishes this by communicating with a process called *VM agent* running inside each VM instance. The process is invoked when the VM boots up and executes commands for various tasks required for managing the VM state.

When the VM agent receives from the local agent the information on the system environment that needs to be provisioned, it first queries the data repository to search for a prebuilt system environment. If found, the agent updates the VM state to the specified state and informs the local agent that the environment is ready. Otherwise, the local agent is responsible for provisioning the desired system environment locally, which means that required components should be built in the VM from a pristine state or from another locally stored system environment.

The *data repository* and the *environment providers* are the sources where *VM agents* look for prebuilt system environments. The data repository and the environment providers will be located in different sites. The provider creates prebuilt system environments, stores both the environments and the update files in its local file server, and register the environments to the data repository. When a query for a specific environment is submitted to the data repository, the URL for the incremental update file located in a file server is returned to the VM agent. Since each test site will run both the *Ede* client and the *environment provider*, the file server containing the update files would be often the one co-located with the *Ede* client.

In the following sections we discuss the structure of each part of *Ede* in more detail.

A. Obtaining Pristine States and Incremental Updates

The procedure for using *Ede* has three phases:

1) Select and "sign" a pristine state of a system environment, and share the state with all participants.
2) Update a pristine system environment and obtain *incremental updates*.
3) Transport the incremental updates to another participant who knows of the same pristine state, and apply the updates to a local copy of the pristine state.

As we discussed in Section III, all participants who want to share system environments have to consent to the same set of pristine states. Thus phase 1 only needs to be executed once, and then copies of all virtual machine images that realize the selected pristine states will be shared by all participants. Afterwards, the sharing and caching activities will require only phase 2 and/or phase 3.

For the software testing scenario aforementioned, the set of components deployed in a system is considered as a system environment. In typical Unix/Linux systems, deploying a component involves copying binary files to the correct directories with proper permissions, creating symbolic links and setting up environment variables.

Based on the *rdiffdir* utility [7] for Unix/Linux, we developed a tool that performs three operations: **sign, update** and **replay**. The **sign** operation is used to set a system state as a *pristine state*. This is done by creating a signature file for all the files under the monitored directories. The signature file describes the state of all monitored files at the binary level, and is shared to all participants together with the virtual machine image that realizes the pristine state. When a participant changes files under in the monitored directories (i.e., the system environment has been updated), the **diff** operation is executed to identify the update information from the pristine state. The incremental updates are encapsulated as *delta* files. Since each sharing party has a copy of a pristine state and its signature file, only *delta* files needs to be transported for other participants to apply the updates. When a participant receives a *delta* file, the participant can execute the **replay** operation to reconstruct the updated system environment by applying the updates contained in the delta file on the pristine system state.

B. Updating Repository

The data repository provides a set of SOAP-based Web services. The users of the repository are automated software testing systems and they will insert and request system environments at run time during test sessions. Therefore the

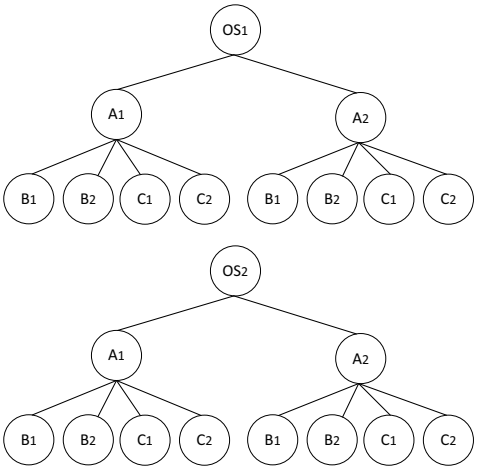

Figure 4. Example Environment Space

repository must be designed to address two major issues: (1) how to store the *delta* files for all available system environments and (2) how to answer the queries efficiently.

As mentioned earlier in this Section, the repository does not store the delta files locally. Instead the repository stores *URL* to the *delta* files registered to the repository. System environment providers are responsible for registering the environments stored in their file systems. Since a tester often plays both the role of environment provider and environment consumer in the testing scenario, the *delta* files would be stored in the file server owned by the tester. That is, each tester will create and store a set of system environments and updates locally and register them to the repository. When other participants later requests the registered environments, the requests will be redirected to the file server owned by the tester. It is also possible that a provider registers an environment to the repository and then deletes it from its local file server due to the limited space. In that case, the participant requesting the environment will get a failure message and will have to rebuild the environment from scratch on a pristine state.

From the model described in Section III, if we have the complete information on all available pristine states and possible updates, the space of all system environments can be determined. For example, consider two pristine states: OS_1 and OS_2, and suppose that we install component A followed by B or C, each of which has two versions. Then, the full environment space can be represented as the forest shown in Figure 4. Each path from a root to a node in the forest represents a feasible system environment. The data repository builds and maintains the forest when environments are registered. The complexity for querying an environment will be $O(log\frac{N}{m})$ for the average cases, where N is the total number of available environments, and m is the total number of available pristine system states.

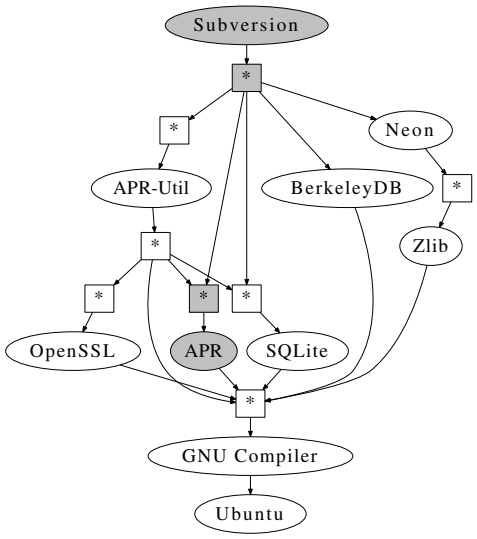

Figure 5. CDG for testing Subversion

Component	Description
Subversion	version control system
Neon	HTTP and WebDAV client library
Zlib	compression library
BerkeleyDB	library for embedded database
APR	supporting library for Apache projects
APR-util	support library for *APR*
SQLite	SQL database engine
Openssl	open source toolkit for SSL/TLS
Gcc	GNU C compiler
Ubuntu	Ubuntu Operating System

Table I
COMPONENTS REQUIRED FOR TESTING SUBVERSION

V. EXPERIMENTS

In this section we evaluate the performance of our infrastructure when it is used for the component-based software testing scenario. We first record the size of *delta* files and the time to obtain and replay *delta* files when updates are applied to the pristine state. We also estimate the time that saved by employing our infrastructure.

A. Virtual Machine Agent Performance

We incrementally build and deploy a set of software components as shown in Figure 5 to a pristine state (an installation of Ubuntu 13.04 server edition), obtain a *delta* file after deploying each component, and apply it to another pristine state. The descriptions on the components in Figure 5 are given in Table I.

The pristine system is hosted as an instance of a VirtualBox virtual machine and we allocated 2048MB of memory for the virtual machine. The size of *delta* files and the times to apply the delta files are shown in Figure 6 and Figure 7.

Figure 6 and Figure 7 clearly show that our environment differencing technique can greatly reduce the size of system environments and enables developers to reconstruct environments required for testing their software with little overhead

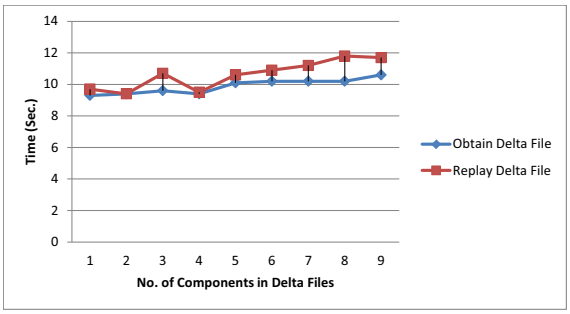

Figure 6. Average Times to Obtain and Replay Delta Files

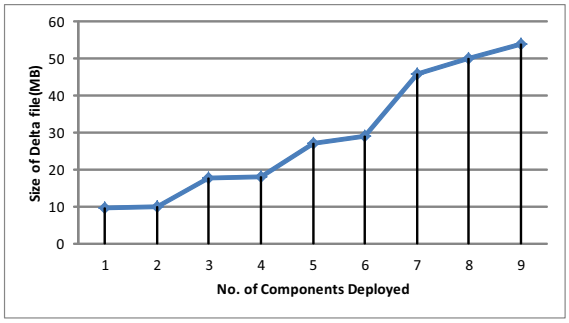

Figure 7. Delta File Sizes

of obtaining and replaying *delta* files.

In Figure 6, we observe that the average time cost to both obtain and replay a *delta* file are around 10 seconds and increasing very slowly when the number of components in a dela file increases. Although not shown in this paper, the times are very small compared to the times to build individual components from source code [2]. Figure 7 shows that the size of a *delta* file does grow linearly as the number of components installed in the system increases, but is still considerably smaller than a typical full virtual machine image, and therefore it is feasible to cache a large number of *delta* files locally and/or also transport them over a wide-area network.

B. Collaborative Testing with Rachet

In order to measure the savings achieved by employing our infrastructure, we recorded total times to build components in the CDG in Figure 8 on virtual machines running on VirtualBox. For each component, we only recorded the time required for building the component itself, assuming that all other components it relies on are available in a system environment – i.e., they are already built and deployed in the virtual machine. In Figure 8, the top-level components are *SVNKit* and *Serf*. *SVNKit* is an Open Source Pure Java Subversion Library, and *Serf* is a high performance C-based HTTP client library. For each component, we monitored all version releases over one year, and each version is tested with existing versions of the components it depends on, and testing of any components that are built upon this new version will be triggered as well. The direct-dependency

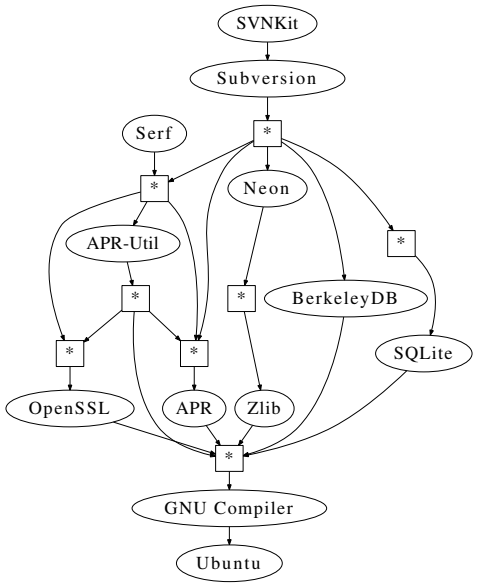

Figure 8. Integrated CDG for Subversion and Serf

coverage (DD coverage [2]) criterion is used to calculate what new configurations introduced by the new component version should be tested. We also include component versions released in the two years before 8/3/2012, so that they are tested together with components released in the one year period. We then feed the times to build the components into a simulator to calculate the total times to build all the components in the CDG with three different cases we now describe.

In the first case, testers of *SVNKit* and *Serf* build their configurations separately within two different *Rachet* sessions. Each *Rachet* session reuses full virtual machine images, so there is a limit on the number of cached system environments. We set the maximum number of environments that can be cached to 16 for this experiment, assuming that a single virtual machine takes 1GB and each *Rachet* node has roughly a 20GB of environment cache space.

In the second case, both testers cache incremental updates instead of full virtual machine images for their *Rachet* sessions. Based on the sizes of *delta* files, as shown in Section V-A, the maximum number of cached *delta* files is set to 1000. Although they use incremental updates, they still test isolated without collaboration. In the third case, the two testers exchange and reuse *delta* files with each other via the data repository. Since *delta* files are small and also multiple data servers are used, we did not restrict the cache size for this case. We then record the total time each tester has to spend on provisioning the system environments required for building and testing their components.

Columns two to four of Table II show the total wall clock time for the testers of the two top-level components to build their test environments that might be realized in a end-user's machine during the selected one year. The last two

Component	Case 1	Case 2	Case 3	Save-1(%)	Save-2(%)
SVNKit	2495128	683246	24378	72.6	99.0
Serf	90812	27020	13742	70.2	84.9
total	2585940	710266	38121	72.5	98.5

Table II
TIMES TO BUILD COMPONENTS (SEC.) AND TIME SAVINGS(%)

columns show the time savings from caching more system environments in the limited space and also from sharing the environments across the testers. We see in Table II that the technique to cache and share delta files for reconstructing system environments boosted the performance of *Rachet* significantly. This is mainly because *Rachet* was able to cache a large number of system environments in the cache space. The environment provisioning time is reduced by 72.6% for *SVNKit*, and 70.2% for *Serf*. When we assume that *Ede* is used to share environments between the testers, we see a very large time savings. 99.0% of the testing time can be saved for *SVNKit*, compared to the case of using *Rachet* without the techniques we have described in this paper. Similarly, we see an 84.9% time savings for *Serf*. It is clear that using *Ede* greatly improves the performance of automatic testing systems by caching many system states and also by sharing test environments between testers.

VI. RELATED WORK

In this work, we focus on enabling a distributed set of users to share and cache virtual machine system states efficiently, by enabling users to only update from the same set of initial states. We employ binary-level system backup and restore tools to obtain incremental changes from the initial states. Using the smaller size of the incremental parts of the virtual machine states, instead of full virtual machine images, makes caching and sharing more scalable. To the best of our knowledge, we are the first group to adapt incremental system backup techniques for this purpose. However, there are studies that address related aspects of the problem.

SnowFlock [8] enables rapid virtual machine cloning for cloud computing. The system forks a large number of virtual machine states efficiently from the same initial instance, and deploys them as individual workers to multiple hosts for cloud computing. Different from *Ede*, forked virtual machines are no longer related to each other, and there is no exchange of states after the initial fork. In addition, the virtual machines deployed to multiple hosts are full VM images, so scalable caching and transport of those image would be infeasible, or very expensive. VMPlants [9] provides the automated configuration and creation of flexible VMs that can be configured once and then instantiated in a homogeneous execution environment. That system focuses more on automatic adaptation to different host environments instead of the guest environments within virtual machines. VMware Workstation 8 and later versions can provide shared virtual machines to multiple users [10], but the machines are hosted and manipulated at a single site, and remotely accessible by the users. None of these systems address the problem of caching and reusing large virtual machine images.

Incremental backup of various system environments is a well developed technique, and there are many such options for Unix/Linux based operating systems. The most common ones are *dump* and *tar*. *dump* provides different levels of backups that save files with variable frequency, and *tar* is widely used for not only backups but also data transport because of its efficiency. There are also well known commercial storage management systems that include incremental backup functionality, such as ADSM [11][12] and Petal [13]. ADSM has the advantage of supporting on-line backup with different levels of consistency, while Petal allows multiple network-connected servers to operate on a pool of physical drives.

Dolstra and Loh have developed a Linux-based operating system distribution called NixOS [14], which has a pure functional approach to system configuration management. When NixOS gets updated, the system grows incrementally and no old configuration gets overwritten. Rebuilding a configuration is deterministic and relatively cheap. Furthermore, Burg and Dolstra deployed NixOS in virtual machines and automated system tests in such VMs for configuration specific testing [15]. QEMU [16] is another open source processor emulator which supports live snapshots of the guest OS it is running. The live snapshots also records only incremental parts of file system updates. Even though there are certain similarities between these work and ours, the NixOS research is focusing on system testing in single configurations and does not address distributed testing sites, and the features of NixOS also cannot be generalized to other systems. QEMU's incremental live snapshot is not generally available to other VM systems either, which cannot support the heterogeneous environment of distributed virtual machines.

VII. CONCLUSION AND FUTURE WORK

This work is driven by the need to reduce the space required to cache virtual machine states and the time to transport virtual machine states across a network, specifically targeting collaborative component-based software testing. In our work, the content of system environments in virtual machines are updated incrementally, and such incremental updates can cost-effectively be cached and shared between testers.

We have modeled a deterministic technique to update system environments from incremental updates, and formally defined the incremental updates on a pristine base system. We have designed and developed an infrastructure called *Ede* to support system environment caching and sharing for automatic software testing systems. *Ede* can reconstruct

system environments required for testing incrementally. We measured the space required to store *delta* files and also the time for obtaining and replaying the *delta* files. We also evaluated the performance of our infrastructure over one-year's data from incrementally testing a set of software components using our *Rachet* automated software build testing system. The experimental results show that *Ede* can create and replay incremental changes to system environments in virtual machines efficiently, and the size of such incremental updates are small enough that they are suitable for large scale caching and transporting across even wide-area networks. When *Rachet* was modified to use *Ede*, the time *Rachet* required for building test environments was significantly reduced.

Our future work includes extending *Ede* to support better *Ede* user authentication and developing methods to use encrypted *delta* files to improve security. We will also integrate *Ede* more tightly into the entire process of collaborative testing of component-based systems, including the testing of *Ede* with different virtualization resources such as public and private cloud computing infrastructure. We also plan to enhance the mechanisms to identify and distribute pristine systems.

ACKNOWLEDGMENTS

This work was partially supported by the US National Science Foundation (ATM-0120950, CCF-0811284, CNS-1205501, CNS-0855055), the National Research Foundation of Korea (NRF-2013010695), and MSIP of Korea (NIPA-2013-H0203-13-1001).

REFERENCES

[1] "Internet Explorer Application Compatibility VPC Image," *http://www.microsoft.com/en-us/download/*, 2014.

[2] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter, "Effective and Scalable Software Compatibility Testing," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA 2008)*, 2008, pp. 63–74.

[3] D. Milojičić, I. Llorente, and R. S. Montero, "OpenNebula: A cloud management tool," *Internet Computing, IEEE*, vol. 15, no. 2, pp. 11–14, 2011.

[4] I. Yoon, A. Sussman, A. Memon, and A. Porter, "Testing component compatibility in evolving configurations," *Information and Software Technology*, vol. 55, no. 2, pp. 445–458, 2013.

[5] ——, "Towards incremental component compatibility testing," in *Proceedings of 14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE-2011)*. ACM, 2011, pp. 119–128.

[6] T. Long, I. Yoon, A. Porter, A. Sussman, and A. Memon, "Overlap and synergy in testing software components across loosely-coupled communities," in *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering (ISSRE 2012)*. Dallas, TX, USA: IEEE Computer Society Press, 2012.

[7] "Duplicity: Encrypted bandwidth-efficient backup using the rsync algorithm," *http://httpd.apache.org*, 2014.

[8] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "Snowflock: Rapid virtual machine cloning for cloud computing," in *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. ACM Press, 2009, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519067

[9] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo, "VMPlants: Providing and managing virtual machine execution environments for grid computing," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC'04)*. IEEE Computer Society Press, 2004. [Online]. Available: http://dx.doi.org/10.1109/SC.2004.67

[10] "Running VMware Workstation as a Server with shared virtual machines," *http://www.vmware.com/products/workstation/*, 2013.

[11] "Distributed Storage Manager (ADSM) C Distributed Data Recovery White Paper," *http://www.storage.ibm.com/storage/software/adsm/adwhddr.htm* .

[12] "Distributed Storage Manager (ADSM) C Frequent Asked Questions," *http://www.storage.ibm.com/storage/software/adsm/adwhddr.htm*.

[13] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. New York, NY, USA: ACM, 1996, pp. 84–92. [Online]. Available: http://doi.acm.org/10.1145/237090.237157

[14] E. Dolstra and A. Löh, "NixOS: A purely functional Linux distribution," *ACM SIGPLAN Notices*, vol. 43, no. 9, pp. 367–378, Sep. 2008. [Online]. Available: http://doi.acm.org/10.1145/1411203.1411255

[15] S. van der Burg and E. Dolstra, "Automating system tests using declarative virtual machines," in *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE 2010)*, 2010, pp. 181–190.

[16] "QEMU Open Source Processor Emulator," *http://wiki.qemu.org/*, 2014.