# DEUX: Autonomic Testing System for Operating System Upgrades

Leon Wu Gail Kaiser Jason Nieh Christian Murphy
Department of Computer Science
Columbia University
New York, NY 10027, USA
{leon, kaiser, nieh, cmurphy}@cs.columbia.edu

## Abstract

*Operating system upgrades and patches sometimes break applications that worked fine on the older version. We present an autonomic approach to testing of OS updates while minimizing downtime, usable without local regression suites or IT expertise.* DEUX *utilizes a dual-layer virtual machine architecture, with lightweight application process checkpoint and resume across OS versions, enabling simultaneous execution of the same applications on both OS versions in different VMs. Inputs provided by ordinary users to the production old version are also fed to the new version. The old OS acts as a pseudo-oracle for the update, and application state is automatically re-cloned to continue testing after any output discrepancies (intercepted at system call level) - all transparently to users. If all differences are deemed inconsequential, then the VM roles are switched with the application state already in place. Our empirical evaluation with both LAMP and standalone applications demonstrates* DEUX*'s efficiency and effectiveness.*

## 1. Introduction

Computer operating system developers launch newer versions of the operating system or patches on a regular basis in order to support some new or improved features, fix some security or stability issues, or simply implement a better or trendy software design. Having encountered many problems such as application incompatibility, application malfunction, and even system crashes caused by operating system upgrades in the past, end-users are getting more and more reluctant to be the first victim of an operating system upgrade. Many experienced corporate system administrators often wait a long time after most initial issues have been fixed and potential problems have surfaced and been analyzed before deployment of a major operating system upgrade in order to reduce the risk and possibility of service disruption [3].

To lessen the problem, software vendors, distributors, and open source developers employ beta testing and package management systems to improve the upgrade quality. However, it is simply impossible for vendors to anticipate and test their upgrades for all the applications and configurations that may be affected by the upgrades at the end-users' machines [10]. Also, package management systems are based on software dependency, which does not take into consideration completely independent third party software that the end-users may be using.

We have therefore designed, implemented and evaluated an approach that focuses on autonomic support for time-consuming and error-prone operating system upgrading and patching activities that typically make applications unavailable to end-users by automatically testing new versions based on normal end-user activities with the old production version (while sandboxing the new version such that it is invisible to those end-users). The prototype implementation of the autonomic testing system, named DEUX, automates testing of operating system updates while minimizing downtime, usable without local regression suites or IT expertise. It also enables fast switchover of the new operating system into production, leaving the older operating system running unnoticeably in parallel if testing needs to continue, without having to install the new operating system twice. An alternative approach would be to place the new version into production immediately, e.g., in the case of urgent security patches, and run the old version as the sandboxed second instance (i.e., to receive the cloned inputs, with output comparison and logging). Then if the new version proves too buggy, the old version is ready to resume production status with minimal loss of users' work. DEUX can also be useful to beta test installations, to isolate the new beta-test version from the old production version that continues to support end-users.

An autonomic testing system for this purpose must satisfy the following requirements. First, it should not require the existing operating system to be modified and currently running applications to be changed. This requirement en-

sures that the existing system and applications will run as usual. Second, the operating system upgrade should be applied to a second sandboxed environment and applications must be tested in this new environment. It would reduce hardware requirements if the sandboxed environment can be created in a virtual machine. Third, it should be easy to compare the application behaviors between the existing system and the sandboxed environment, and the results should be logged and reports should be generated. Fourth, when discrepancies or errors happen in the sandboxed environment, the system state must be saved and sandboxed environment stopped and then restarted so that it restarts from the existing system's current state. Finally, the overhead and efficiency of the testing system should be acceptable.

To address these requirements, our approach includes a virtual machine environment, a checkpoint and restart mechanism, autonomic and simultaneous execution of user applications, and logging and reporting functions. Our approach also takes advantage of "pseudo-oracle" testing: identical user inputs are supplied to two copies of the same application, one running on the old operating system and one running on the new, and the resulting outputs are compared [12]. In general, if the results are semantically different, then a defect must exist in the operating system and/or the application. We cannot yet address intentionally nondeterministic applications. The applications in both environments run in parallel transparently to the user, and the virtual machine environment can be constructed without requiring new hardware. Also, both running systems can take checkpoints and save their state information as two separate physical files, and either system can then be restarted using the checkpoint snapshot file the other system generated. Furthermore, the input, output and other results can be logged and used to generate reports. After testing for a sufficient time as determined by the administrators or computer owners, the decision of committing to the operating system upgrade or staying with the older one can be made. The end-users of the system do not even need to know about the testing or the cloned applications running in the second virtual machine.

The rest of paper is organized as follows. Section 2 presents our motivation, limitations of existing approaches, and overview of our approach. Section 3 depicts the design including architecture and key components: the virtual machine environment, checkpoint and restart mechanisms, autonomic and simultaneous execution, and logging and reporting. Section 4 describes the implementation, and Section 5 evaluates the system with test cases, experimental results and performance metrics. Section 6 analyzes the related work, and lastly Section 7 concludes and outlines future work.

## 2. Motivation and Approaches

### 2.1. Motivation

Operating system failures and incompatibilities are some of the major causes of application malfunction and system downtime. As is the case with other software, operating systems and operating system upgrades come with defects. According to prior research, defects remain in the Linux kernel an average of 1.8 years before being fixed [8].

Large corporations and government agencies spend a huge amount of resources each year dealing with operating system upgrades and maintenance [9]. System administrators in these organizations often set up sophisticated lab environments to test operating system upgrades for a long period of time before actual roll-out to make sure the upgrade has minimal impact on in-house applications.

In some small or medium companies or organizations with limited resources, however, upgrading the operating system can be a daunting task. For instance, a small nonprofit charity organization with just a few non-technical staff may only have one computer to manage all their fundraising activities, human resources, and accounting records. If a new upgrade is available and urgent, they may go ahead with the installation, but a defect in the upgrade may render their system unstable and break some applications. The higher risk is that the system crashes or applications can't be used at all. In that case, it is often needed to revert to an old version of the OS, which might be a challenging task for non-technical personnel. A user-friendly and effective autonomic testing system that does not require extra hardware resources and sophisticated computer knowledge to operate would solve the problem and alleviate people's fear of upgrading their operating system. Such a system would swap in the new operating system version when all seems to be working correctly, so the new OS version has to be installed only once - and there is essentially no application downtime.

### 2.2. Limitations of Existing Approaches

There are certain common limitations in existing beta testing and package management systems approaches we studied.

The first limitation is that the existing approaches are not comprehensive and do not cover end-users' environments and the applications of their specific interest. For example, although beta testing before an operating system release and package management systems improve the quality of the release, they do not provide a comprehensive testing mechanism to encompass end-users' specific environments and applications. Some users may have proprietary or third party software applications to which only they have access.

Package management systems will not ensure that these applications work well after an operating system upgrade.

The second limitation is that the existing approaches require extra resources such as new computer hardware. For a corporate IT department, it is very common to have some extra idle machines in a lab for testing the new operating system or operating system upgrade. A more advanced data center operator usually has a high-end deployment and testing management system along with different kinds of running environments to perform testing. These kinds of testing labs are expensive to set up and operate, and small or medium businesses and home users may find the approach beyond their budget and time. They often trust the operating system developers blindly and install the operating system upgrade at their own risk, without any testing.

The third limitation is that the existing approaches are usually manual. They require human operators to test the applications in the newly upgraded operating system. In order to have better testing results, the human operators have to first record the functionalities commonly used in the old operating system and perform functionality testing in the new operating system. Then they have to perform other testing such as compatibility, performance and security testing manually. Some big data centers or corporations might have specialized software to automate these tasks. But these software are usually very expensive and out of reach of normal users.

### 2.3. Our Approach

We present an automated testing system named DEUX that provides autonomic support for local testing of operating system upgrades while minimizing application downtime, intended for small-scale IT operations without a dedicated IT staff or the resources to construct a local regression test suite. DEUX utilizes an architecture based on a virtual machine (VM) environment, with a second level of lighter-weight virtualization supporting the checkpointing, migration, and resuming of application process across minor operating system versions, enabling simultaneous execution of the same application on both old and new versions of the OS in different VMs. The input provided by ordinary users, continuing to use the production application on the old OS version, is fed to the new version. Any output discrepancies are logged, with the old version effectively acting as a pseudo-oracle for the new version. The application state is then automatically cloned again to continue testing; this is all transparent from the users' perspective. If after some period of usage all output discrepancies between the two versions are deemed inconsequential (or if the new version fixes known flaws in the old version), then the VM roles are switched - putting the new OS version into production with the application state already in place.

Our approach overcomes the above mentioned limitations. First of all, our approach gives the end-users the tools and flexibility to run tests on the applications they choose and the functionality they choose. In this way, there is no need to worry about the testing being comprehensive because end-users can try all the applications as they normally use the computer. For other applications and functionality they do not use, testing would not be required.

Second, our approach takes advantage of the virtual machine environment. The parallel testing of the new operating system along with the old operating system can run inside the same computer. End-users do not need to acquire extra hardware for testing the operating system upgrade. The virtual machine environment also provides isolation and protection of the applications running in the production mode in the existing operating system from the applications running in the testing mode in the new operating system.

Third, our approach automates the testing using autonomic and simultaneous parallel execution. The end-users do not need to redo the actions of providing input for the application instances running inside the new operating system, which is in turn running inside the virtual machine. After launching, the end-users can perform their usual tasks in the existing operating system. The applications would run in parallel inside the virtual machine and accept user input transparently.

Fourth, our approach does not require special knowledge of software testing. The end-users don't even know the autonomic testing is happening. DEUX requires that administrators or computer owners be able to look at the discrepancy reports and determine whether any differences matter or not, but this does not affect the end-users.

## 3. Design

### 3.1. Overview

As illustrated in Figure 1, DEUX consists of two virtual machines, one shared directory, a mutual client, and a host machine. The current prototype system is implemented for the Linux environment.

The two virtual machines, VM1 and VM2, run the old operating system, namely OS1, and new operating system, namely OS2, respectively. Inside each virtual machine, there is a PrOcess Domain (POD). The POD architecture was originally designed and implemented by Columbia University researchers using Zap, a system for migrating computing environments [23]. A POD provides a group of processes with a private namespace that presents the process group with the same virtualized view of the system. This virtualized view associates virtual identifiers with operating system resources such as process identifiers and net-

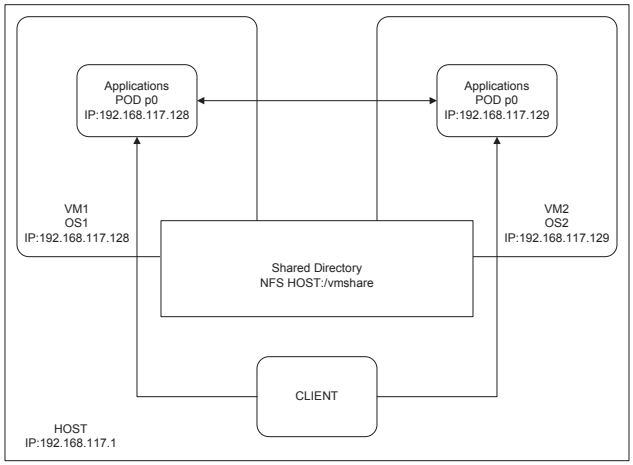


**Figure 1.** DEUX **architecture**

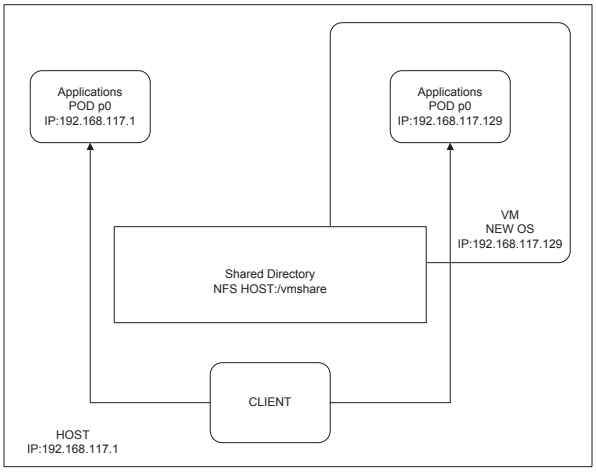


**Figure 2. Alternate** DEUX **architecture**

work addresses. This decouples processes in a POD from dependencies on the host operating system and from other processes in the system [23]. The user applications run as processes inside each POD.

Why do we need both PODs and VMs? The main point is that PODs support migration across different OS versions [23], while doing a checkpoint of a VM would include the *entire* OS and thus there would be no way to switch OS versions. The POD, in contrast, takes only the applications and a thin virtualization layer, and can be resumed on a different OS version.

The shared directory, *e.g.* Host:/vmshare in the diagram, is for easier file sharing between the host and the two virtual machines. One type of important file for sharing is the checkpoint snapshot file. VM2 needs to access the checkpoint snapshot files that VM1 generated in order to quickly restart its state to match up the environment of VM1. Some read-only files can also be easily shared between the host machine and two virtual machines.

The client is for the administration of the testing, so that the end-user does not notice the existence of the testing and applications running inside the POD of VM2. It has access to the host machine and two virtual machines. When the client gives any input to VM1, the input is also replicated to VM2. In this way, applications running inside OS1 and OS2 in different virtual machines are given synchronized inputs to enable comparison of the respective outputs.

An alternate architecture of DEUX is illustrated in Figure 2. This architecture is a simplified version of the architecture previously described. The difference is that this architecture uses only one virtual machine. The host machine runs the old operating system OS1, while the single virtual machine runs the new operating system OS2. Although it is simpler, it does not provide the flexibility of comparing operating systems besides the one being used by the host machine. The absence of the first virtual machine also makes it impossible to swap production and testing VM roles. It means if the testing of the new OS goes well, there is no immediate switchover that can be done to make the new OS the production OS, thus reducing the downtime and OS upgrade installation time.

Of course it is possible that a defect on the OS may prevent DEUX from running in the first place. If the new OS has a defect that prevents DEUX from operating, then we know it has a defect and don't have to do anything further (except maybe log the defect and report it to the vendor). The testing ends here. If DEUX runs fine, then we need to try each of the user applications, to see if any of them triggers a defect. If none of them causes a defect, then we put the new OS version in production. So defects that prevent DEUX from running aren't an issue.

### 3.2. Virtual Machine Environment

The use of a virtual machine provides a key virtual execution environment for the new and old operating systems in DEUX. There are several advantages of utilizing virtual machines in the design. One obvious advantage is that a virtual machine reduces the hardware resource cost. The virtual machines are easy to be integrated and different components can communicate with each other in various ways such as directory sharing, SCP or SSH. Furthermore, the virtual machine provides better isolation and protection because the applications running inside are limited to the resources and abstractions provided by the virtual machine.

There are many different virtual machine products avail-

able. We used VMware Server [28] for Linux in the prototype implementation. In order to optimize the system performance in the POD, DEUX employs a copy-on-write (COW) technique as described in [19]. From the virtual file system perspective, directories and files are categorized as either Read Only or Read/Write. The Read Only directories and files are shared and linked each time the new POD is created, while the Read/Write directories and files are POD-specific with each POD having its own Read/Write directories to store specific information for that POD.

### 3.3. Checkpoint and Restart

A checkpoint is a mechanism to save a snapshot of state information of the running process or process group into a persistent format, so that it is possible to later revive the session. The checkpoint does not require complete termination of the current process or process group: it can either pause or kill the process or group. The restart is a mechanism of reviving a saved session with state and processes in either the same environment or a different environment as if nothing has happened in between, using the checkpoint snapshot file.

The prototype implementation of DEUX uses the checkpoint and restart technique employed by DejaView [19]. DejaView is the new improved version of Zap mentioned previously. In the case of taking a checkpoint, the session is quiesced and all its processes are forced into a stopped state, to ensure that the saved state is globally consistent across all processes in the session. Then the execution state of the virtual execution environment and all processes is saved. Also, a file system snapshot is taken to provide a version of the file system consistent with the checkpointed process state.

Figure 3 illustrates DEUX's checkpoint and restart algorithm. As a typical scenario, first, the application has to be started in VM1's POD, and then it is paused and checkpointed. Its checkpoint snapshot file is saved to a shared directory Host:/VMShare with file name vm1-ck0. VM2 can then start a POD using snapshot file vm1-ck0; that is, the application process is migrated to VM2. At the same time, the POD in VM1 is resumed. This is the first clone phase. The actions are also being logged into the database during this process. After that initiation stage, the user can begin to interact with the application as it runs in VM1. When an input A is given to VM1 (OS1), it is passed to VM2 (OS2); VM1 produces output A-1, and VM2 produces A-2. A-1 is compared with A-2: if they are identical (in the case of deterministic testing), then DEUX does nothing and continues to the next test.

However, if A-1 is different from A-2, first, the results are logged into the database, then both PODs running in VM1 and VM2 are checkpointed and the snapshot files are saved as vm1-ck1 (from VM1) and vm2-ck1 (from VM2)

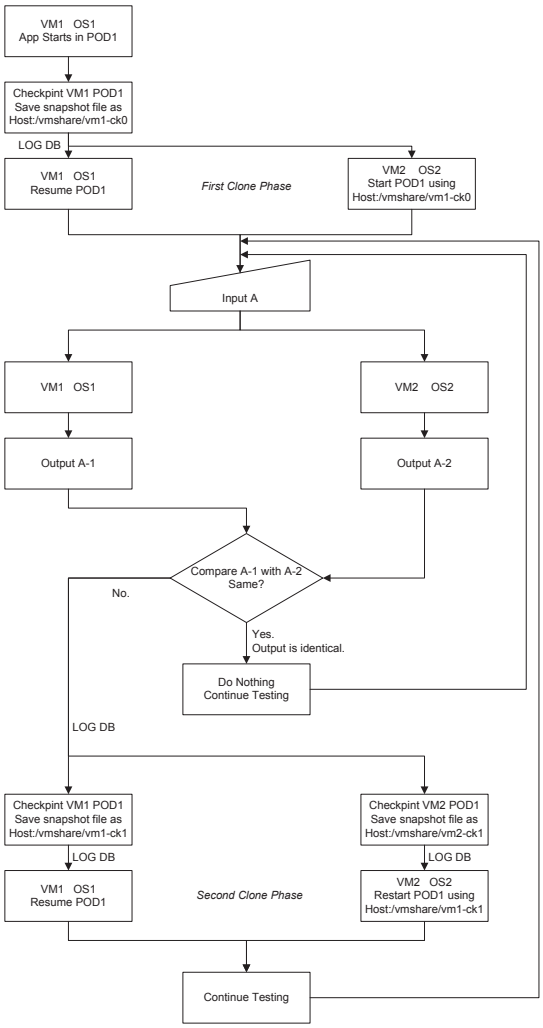


**Figure 3. Checkpoint and restart algorithm**

respectively. The second clone phase begins when the POD inside VM1 is resumed. At the same time, the vm1-ck1 snapshot file previously saved from VM1 is used to restart the POD inside VM2, so that after restart, both PODs in VM1 and VM2 are starting from the identical state. The actions are also logged into the database. Finally, the testing continues until the satisfactory conclusion is reached. In the case that the results are non-deterministic, human judgment is needed to decide if application running on OS2 is actually deviated from application running on OS1. Future work might include semi-automated analysis tool that is able to aid human decision.

### 3.4. Autonomic and Simultaneous Execution

Autonomic execution means DEUX is capable of self-execution and self-management without human intervention. Simultaneous execution means the applications run inside the two virtual machines in parallel and do not require the computer users to give input twice. The users only need to focus on the applications running on the existing operating system, which is inside the POD of VM1 (OS1). The applications running inside the POD of VM2 with the newer operating system OS2 would run in a mirrored and autonomic fashion.

The passing of user input (or action) from POD of VM1 to POD of VM2 and vice versa is achieved via user interface (UI) monitor daemon as illustrated in Figure 4. The monitor daemon runs as a persistent service inside both PODs. Whenever the user input is detected in one POD, the actions can be synchronized to the other POD by passing through the client, who administrates the testing. The client acts as a console to control the direction of the input synchronization and also can initiate actions such as checkpoint and restart.

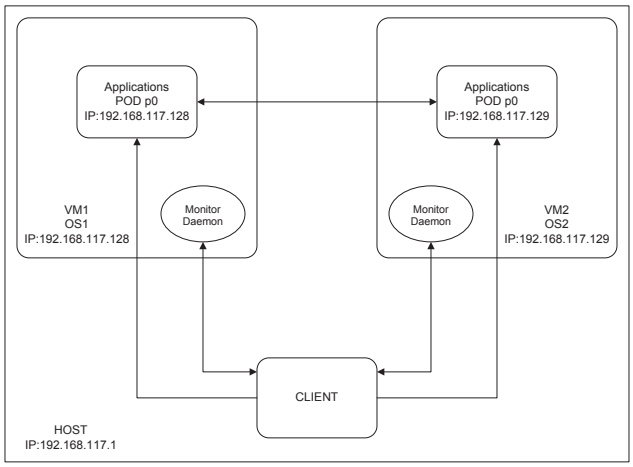


**Figure 4. UI monitor daemon**

It is possible to utilize THINC visual display architecture [2] and Dejaview to record display and contextual information by capturing the name and type of the application that generated the text, window focus, special properties about the text, and all text that is displayed on the screen [19]. In our prototype implementation, the application output discrepancies require intervention from administrators or computer owners, but not end-users.

One advantage of autonomic and simultaneous execution is ease of testing. End-users do not need special technical knowledge or extra time in designing different test cases. They can just run the applications as they normally use the computer. DEUX would provide the necessary testing mechanism along the way.

Another advantage is the better accuracy of the testing because the application running in the old version of the OS is used as a pseudo-oracle for the application running in the new version. Testing is time sensitive. Simultaneous processing of input, output and other functions are crucial for determining the quality and performance of the applications and the underlying operating systems.

### 3.5. Logging and Reporting

In order to keep records of the testing procedures, results, checkpoint and restart times, and error messages, DEUX provides logging and reporting functionality by saving the testing information into a lightweight database running on the host machine.

As described in Table 1, the main tables of the database include OS, Application, Test, Checkpoint, and Restart.

**Table 1. Database tables**

| Table Name | Description |
|---|---|
| OS | OS version information |
| Application | Applications being tested |
| Test | Testing sequence with timestamp |
| Checkpoint | Checkpoint file, source VM and time |
| Restart | Restart snapshot used, VM and time |

The entire process is meant to be transparent to the end-users, but the computer owners or administrators would need to look at the application along the way to get an idea when to end testing. Alerts could be generated after some discrepancies have occurred, however, to tell computer owners or administrators to look at the database. They can review the historic information logged in the database and generate reports based on the data saved in order to make better decisions on whether the operating system upgrade should be performed.

## 4. Prototype Implementation

The prototype DEUX system is implemented on an HP Proliant DL 360 G3 server. The server has a single Intel Xeon 3GHz CPU, 4GB RAM, and a 36GB SCSI Hard Drive.

The host operating system is Ubuntu Linux version 7.10 (Gutsy Gibbon). The virtual machine environment is VMware Server version 1.0.3 for Linux. The checkpoint and restart software is DejaView (with version April 2008) [19] and Zap (with version February 2007) [23].

The shared directory is set up using an NFS shared folder /vmshare on the host machine and Read/Write access from VM1 and VM2 is enabled.

The programming languages employed include C, shell scripts and SQL. DEUX has around 20K lines of new code on top of the software employed.

## 5. Evaluation

We evaluated the DEUX system using different test cases and performance metrics. In all our test environments, OS1 is the old operating system on which applications work correctly, and OS2 is the upgraded operating system that we want to test. In our prototype implementation, the operating system for the host machine is Ubuntu Linux version 7.10 with Linux Kernel version 2.6.22. OS1 is Debian Linux with Kernel version 2.6.11, and OS2 is a higher version, 2.6.12.

The first test case is the testing of a LAMP (Linux, Apache, MySQL, PHP/Python/Perl) server application. We tested DEUX using one specific example: phpBB, which is an online bulletin board application using LAMP technologies. The version of phpBB tested is 2.0.22-2 (Debian). The second test case is the testing of a standalone application: we tested DEUX using Mozilla Firefox. The version of Firefox tested is 1.5.0.3.

The performance metrics we employed include resource consumption such as memory and CPU usage, time latency because of checkpoint and restart, and hard drive storage requirements.

### 5.1. Test LAMP Server Application phpBB

phpBB is a popular open source LAMP server application that enables Internet users to post and share messages in an online forum environment and also provides management functionality for administrators.

In order to better resemble end-user activities in the real world, we designed and categorized our test cases into different functional tasks: user login/logout, posting a message, replying to a message, editing a profile, administrator login/logout, configuring the forum settings, user management, sending group email, backup and restore, *etc.* These are representative of what real end-users would actually do in their regular activities when phpBB is deployed in the field.

In all test cases, virtual machine VM1 runs operating system OS1 and virtual machine VM2 runs operating system OS2. The standard testing procedure is as follows: first, phpBB is started in the POD of VM1. Then VM1 is paused and checkpointed. The POD of VM2 is started using the checkpoint snapshot file saved by VM1. At the same time, VM1 is resumed. Second, end-users perform functional tasks as described above. When phpBB running on OS2 has a different output than phpBB running on OS1, or phpBB running on OS2 has an error message while phpBB running

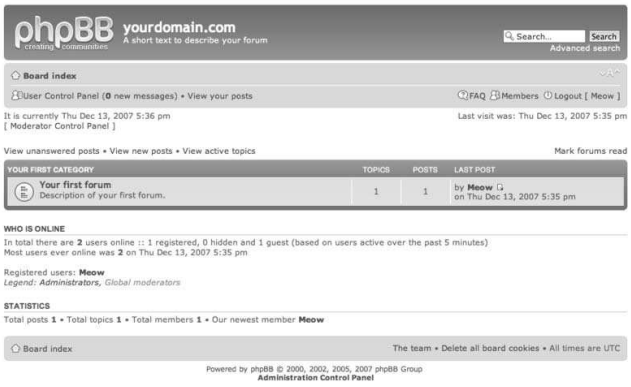


**Figure 5. phpBB**

on OS1 has no error, the system performs a checkpoint of the POD on VM1 to vm1-ck1 and a checkpoint of the POD on VM2 to vm2-ck2. Both vm1-ck1 and vm2-ck2 are saved to the shared directory /vmshare of the host machine. Actions and checkpoint-related information are also logged to the database. The POD in VM1 is resumed and the POD in VM2 is restarted using vm1-ck1 simultaneously. Finally, end-users continue to the next test case and run the tests until all their normal activities in using the application are covered.

### 5.2. Test Standalone Application Mozilla Firefox

Mozilla Firefox is a popular open source cross-platform Internet browser. Along with traditional web browsing functionality, Firefox also provides a built-in news feed reader that supports RSS (Really Simple Syndication), an XML-based data feed format. Furthermore, Firefox is a pioneer in incorporating a search tool box into the user-friendly interface.


**Figure 6. Mozilla Firefox**

Web browsers including Firefox provide simple and commonly used functionalities such as web browsing, news feed reading, bookmarking favorite web pages, *etc.* Since there are not many different functional tasks, we developed test cases covering the different components of the application itself, such as tab browsing, opening and saving files of different formats, changing preferences, searching, creating bookmarks, *etc.* These are normal user activities in the real world.

Similar to the testing of phpBB, in all test cases, virtual machine VM1 runs operating system OS1 and virtual machine VM2 runs operating system OS2. The standard testing procedures are the same for those of phpBB, as described above.

### 5.3. Experimental Results and Analysis

In the testing of phpBB as an example of a server application, we identified one defect, which is related to the email functionality. One manifestation of this defect occurs when the administrator logs in, goes to the Administration Panel, and tries to compose and send a group email using the Mass Email form. The error message was "General Error Failed sending email :: PHP :: DEBUG MODE Line : 234 File : emailer.php". As reported in the phpBB Bug tracker, this is a known defect [25] that only occurred in VM2/OS2, but did not occur in VM1/OS1. This defect also occurred on other occasions when an email was supposed to be generated and sent, such as right after registration, posting a message to a bulletin board, or sending a private message. After further investigation, we verified that the defect is indeed caused by the operating system upgrade. The direct cause was the upgrade of the /usr/sbin/sendmail program. Sendmail is a mail transfer agent responsible for handling mail in most variants of UNIX operating systems. It is a popular target for network intruders. Due to the proliferation of the Internet security breaches, Sendmail is updated frequently and the operating system upgrade often comes with a newer version of Sendmail. In this case, DEUX was able to detect a defect that only revealed itself when using the new version of the OS.

In the testing of Mozilla Firefox as an example of standalone application, we also identified one defect that appeared in VM2/OS2 but did not occur in VM1/OS1. When Firefox displayed a web page with a Flash plugin for a given period of time, Firefox suddenly crashed and exited. Further study of the problem confirmed that the operating system upgrade in OS2 contains a newer library of libnss3, which caused Firefox to run in an unstable condition. The libnss3 library and its newer generations are network security service libraries that support security-enabled client/server applications. In this test, the conflict between Firefox and the newer version of the OS library caused Firefox to crash, and this defect was detected using DEUX. Further investigation shows that this is a confirmed known bug of Firefox [22].

On the other hand, it is possible that OS2 might fix defects in OS1, and so it would be *expected* that the results are different. One such case in the testing of Mozilla Firefox showed that the newer operating system OS2 fixed a defect that only appeared in the old operating system OS1. This defect is related to the mime-support system package. The newer version of the mime-support package includes support for more and newer file formats and file name extensions. In this case, DEUX proved that OS2 has advantage over OS1 and the user should upgrade to OS2 if there are no other issues.

The above experimental results show that the DEUX system is effective in helping the end-users to test the operating system upgrade with no additional operational burden on their part. DEUX is not only able to detect defects that only appear in the newer operating system, but it can also show defects that have been fixed by the operating system upgrades.

### 5.4. Performance Metrics and Evaluation

To better evaluate the DEUX system, we also measured the system performance using metrics of resource consumption such as memory usage, CPU consumption, time latency due to taking checkpoints and restarting, and hard drive storage usage.

**Table 2. Memory usage**

| Services | Memory Used (MB) |
|---|---|
| OS | 1328 |
| OS+App running on host | 1354 |
| OS+single VM | 1449 |
| OS+two VMs | 1589 |
| OS+DEUX (two VMs+two PODs) | 2371 |
| OS+DEUX+App running in PODs | 2641 |

**Table 3. CPU consumption**

| Services | CPU used (in %) |
|---|---|
| OS | 3.5 |
| OS+App running on host | 5.3 |
| OS+single VM | 6.1 |
| OS+two VMs | 10.2 |
| OS+DEUX (two VMs+two PODs) | 11.3 |
| OS+DEUX+App running in PODs | 30.5 |

In Tables 2 and 3, the application used for measurement is the Mozilla Firefox web browser. DEUX includes two

VMs, each running a POD. The Linux command used for capturing memory usage information is ($ free -t -m). The system utility used for getting CPU consumption information is ($ top) and ($mpstat -P ALL). The CPU usages are volatile with some spike during application startup or shutdown. The numbers in the Table 3 are average numbers of five trials.

Table 2 shows that DEUX adds around 782MB of memory usage on top of the VMs. This memory is mostly consumed by the PODs, inside which the user applications are running. For our testing system with 4GB memory, the memory usage of DEUX is acceptable. Table 3 shows that the CPU usage of the PODs themselves is around 1.1% of the total CPU. Given the fact that CPU usages fluctuates drastically among processes, this number is negligible.

In terms of hard drive storage usage, each of the VMs in the prototype implementation occupies around 8.5GB disk space mainly because they are self-contained virtual machines. The disk space usage varies based on the applications that need to run and sizes of the user files.

**Table 4. Checkpoint time**

| Command | Time (in seconds) |
|---|---|
| CK empty POD | Real 0.678 User 0.000 Sys 0.013 |
| CK POD with browser running | Real 1.154 User 0.000 Sys 0.683 |
| CK POD with Emacs running | Real 0.969 User 0.000 Sys 0.568 |

**Table 5. Restart time**

| Command | Time (in seconds) |
|---|---|
| RS empty POD | Real 0.391 User 0.000 Sys 0.016 |
| RS POD with browser running | Real 0.450 User 0.000 Sys 0.013 |
| RS POD with Emacs running | Real 0.502 User 0.000 Sys 0.027 |

We also used the system command ($ time command) to obtain the timing statistics. The results are shown in Table 4 and 5. In the tables, Real means the elapsed real time between invocation and termination, User means the user CPU time, and Sys means the system CPU time. From the tables, the checkpoint and restart time are mostly in the range of less than one second.

Laaden *et al.* measured the detailed latency of the checkpoint and restarts, using the same mechanism DEUX uses, when applied to different applications [19]. According to their results, the latency of taking a checkpoint is less than a few milliseconds, and the latency of performing a restart is less than a few seconds. The latencies are short comparing to functional operations of a lot of user applications. And they do not happen very often. Thus, the end-users would not see the latencies caused by DEUX as an issue.

The above results show the DEUX system is efficient and does not consume significant system resources.

## 6. Related Work

In [26], Qin *et al.* proposed a technique to rollback the program to a recent checkpoint upon a software failure, and then re-execute the program in a modified environment in order to quickly recover programs from many types of software defects. In their implementation, the checkpoint and rollback features were also used. However, there are fundamental differences between their approach and DEUX. First of all, the purpose of the system is different. Qin *et al.* aimed to solve the problem of recovering programs while removing the so-called "allergens", which are the cause of the defects. The checkpoint and rollback mechanisms are used as a way to revive programs. DEUX's goal is to provide an intuitive and effective end-user tool for non-technical users to test the operating system upgrade in their context. The checkpoint and rollback technique used by DEUX is for pausing, stopping and synchronizing the PODs, inside which user applications run as processes.

In [10], Crameri *et al.* proposed Mirage, a distributed framework for integrating upgrade deployment, user-machine testing, and problem reporting into the overall upgrade development process. Their work was motivated by the results of a survey of 50 system administrators. Their focus is on how to better design a deployment system with various different subsystems. On the other hand, DEUX does not aim to improve the deployment or even the operating system. DEUX specifically aims to help the end-users, not the developers or vendors of the operating systems, although they can benefit from better user feedback or they can use DEUX to test the operating system upgrade themselves.

In [30], Yang *et al.* described EXPLODE, a system to systematically check real storage systems for errors. EXPLODE uses checkpoint and restore states to explore and exhaust the choices for one choice point in storage checking. There are at least three differences between EXPLODE and DEUX. First, EXPLODE uses a comprehensive, heavyweight formal verification technique based on model checking to make its checking more systematic. DEUX does not use any formal verification technique in the design. Second, during checkpoint and restart, EXPLODE uses computation rather than copying to recreate states; on the other hand, DEUX takes advantage of the Copy-on-Write file system migration and the states are being saved and exchanged through physical checkpoint snapshot files. Third, EXPLODE and DEUX aim to solve different problems with EXPLODE focusing on finding serious storage system errors

while DEUX provides an automated testing system for operating system upgrades.

In [27], Su *et al.* proposed a configuration management tool named AutoBash that uses operating system causality analysis and OS-level speculative execution to try possible configuration actions, examine their effects, and roll them back when necessary. Its goal is to automate the search for solution of the configuration problem. DEUX does not aim to find a solution of the OS upgrade problems. Instead, DEUX provides the techniques for identifying the issues caused by the operating system upgrades. Furthermore, their system architecture and fundamental methods are different. AutoBash uses causality analysis, search and speculative execution. While DEUX uses lightweight POD running in virtual machines, checkpoint, restart, logging, and autonomic and simultaneous execution.

## 7. Conclusion and Future Work

DEUX provides a new approach that may be applicable to other software quality assurance implementations as well. One possible direction for future work would be incorporating a copy-on-write database into the design of DEUX. The prototype implementation described in this paper does not handle Read Only and Read/Write data of the user application databases separately. In practice, this might require more storage space and be less efficient, however. Further investigation may also look into better ways of automating the process and reducing the required human interventions, such as developing a semi-automated analysis tool for checking the discrepancies of applications. Another future work may include automated construction of local regression test suites from normal user activities, with recording of production inputs and outputs, to enable testing of new versions "offline", e.g., while the production version is idle. There is also possible future work for researchers who are interested in applying similar technologies to Windows or other operating systems.

In this paper, we have presented DEUX, a technology that automates operating system testing using normal user activities as part of its pseudo-oracle approach, while minimizing application downtime caused by operating system upgrade and patching. Our contribution is an approach and a prototype implementation that enables ordinary users to transparently test software applications of their own interest in the new operating system in parallel with the existing operating system transparently, without requiring the end-user to be technology savvy or have knowledge of software testing. The empirical evaluation of the system using test cases of different applications and performance metrics shows that DEUX is efficient and effective.

## 8. Acknowledgments

The authors thank Oren Laadan and Shaya Potter for their assistance. Wu, Kaiser, and Murphy are members of the Programming Systems Laboratory, funded in part by NSF CNS-0717544, CNS-0627473, CNS-0426623 and EIA-0202063, and NIH 1 U54 CA121852-01A1. Nieh is a member of the Network Computing Laboratory, funded in part by NSF CNS-0717544 and CNS-0426623.

## References

[1] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: How to upgrade distributed systems. In *In Ninth Workshop on Hot Topic in Operating Systems (HotOS-IX*, pages 43–48, 2003.

[2] R. A. Baratto, L. N. Kim, and J. Nieh. Thinc: a virtual display architecture for thin-client computing. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 277–290, Brighton, United Kingdom, 2005. ACM.

[3] C. Barker. Gartner: Ignore vista until 2008. available at `http://news.cnet.com/`, 2005.

[4] A. Baumann, G. Heiser, J. Appavoo, D. D. Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 32–32, Anaheim, CA, 2005. USENIX Association.

[5] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including os and network aspects. In *HASE '01: The 6th IEEE International Symposium on High-Assurance Systems Engineering*, pages 95–105. IEEE Computer Society, 2001.

[6] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot a technique for cheap recovery. In *OSDI 04: 6th Symposium on Operating Systems Design and Implementation*, pages 31–44, San Francisco, CA, USA, December 2004. USENIX Association.

[7] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. Polus: A powerful live updating system. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 271–281. IEEE Computer Society, 2007.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. *SIGOPS Oper. Syst. Rev.*, 35(5):73–88, 2001.

[9] A. L. Couch, N. Wu, and H. Susanto. Toward a cost model for system administration. In *19th Large Installation System Administration Conference (LISA 05)*, pages 125–141, San Diego, CA, 2005. USENIX Association.

[10] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 221–236, Stevenson, Washington, USA, 2007. ACM.

[11] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented

programs. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 50–60, London, United Kingdom, 2007. ACM.

[12] M. D. Davis and E. J. Weyuker. Pseudo-oracles for non-testable programs. In *Proc. of the ACM '81 Conference*, pages 254–257, 1981.

[13] R. Griffith and G. Kaiser. A runtime adaptation framework for native c and bytecode applications. *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pages 93–104, June 2006.

[14] P. Gross, S. Gupta, G. Kaiser, G. S. Kc, and J. J. Parekh. An Active Events Model for Systems Monitoring. In *Working Conference on Complex and Dynamic Systems Architectures*, 2001.

[15] G. Kaiser, P. Gross, G. S. Kc, J. J. Parekh, and G. Valetto. An Approach to Autonomizing Legacy Systems. In *Workshop on Self-Healing, Adaptive and Self-MANaged Systems*, 2002.

[16] G. Kaiser, J. J. Parekh, P. Gross, and G. Valetto. Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems. In *Autonomic Computing Workshop*, 2003.

[17] A. D. Keromytis, J. J. Parekh, P. Gross, G. Kaiser, V. Misra, J. Nieh, D. Rubenstein, and S. J. Stolfo. A Holistic Approach to Service Survivability. In *ACM Workshop on Survivable and Self-Regenerative Systems*, 2003.

[18] N. P. Kropp, P. J. K. Jr., and D. P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Symposium on Fault-Tolerant Computing*, pages 230–239, 1998.

[19] O. Laadan, R. A. Baratto, D. B. Phung, S. Potter, and J. Nieh. Dejaview: a personal virtual computer recorder. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 279–292, Stevenson, Washington, USA, 2007. ACM.

[20] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI 04: 6th Symposium on Operating Systems Design and Implementation*, pages 289–302, San Francisco, CA, USA, December 2004. USENIX Association.

[21] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel test generation and execution with korat. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 135–144, Dubrovnik, Croatia, 2007. ACM.

[22] Mozilla. Bugzilla@mozilla. available at `https://bugzilla.mozilla.org`, 2008.

[23] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *OSDI 02: 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, USA, December 2002. USENIX Association.

[24] J. J. Parekh, G. Kaiser, P. Gross, and G. Valetto. Retrofitting Autonomic Capabilities onto Legacy Systems. *Journal on Cluster Computing*, 9(2):141–159, 2006.

[25] phpBB Group. phpbb bug tracker. available at `http://www.phpbb.com/bugs`, 2008.

[26] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 235–248, Brighton, United Kingdom, 2005. ACM.

[27] Y.-Y. Su, M. Attariyan, and J. Flinn. Autobash: improving configuration management with operating system causality analysis. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 237–250, Stevenson, Washington, USA, 2007. ACM.

[28] VMware. Vmware products. available at `http://www.vmware.com/products`, 2008.

[29] L. Wu, G. Kaiser, J. Nieh, and C. Murphy. Deux: Autonomic testing system for operating system upgrade. Technical Report CUCS-037-08, Department of Computer Science, Columbia University, 2008.

[30] J. Yang, C. Sar, and D. Engler. Explode: A lightweight, general system for finding serious storage system errors. In *OSDI 06: 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 131–146, Seattle, WA, USA, November 2006. USENIX Association.

[31] J. Yang, P. Twohey, and D. Engler. Using model checking to find serious file system errors. In *OSDI 04: 6th Symposium on Operating Systems Design and Implementation*, pages 273–288, San Francisco, CA, USA, December 2004. USENIX Association.

[32] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas. iwatcher: Efficient architectural support for software debugging. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 224, München, Germany, 2004. IEEE Computer Society.