Implementing a parallel world model using Linux containers for efficient system administration

(Paper)

Yasushi Shinjo and Wataru Ishida
Department of Computer Science
University of Tsukuba
1-1-1 Tennoudai, Tsukuba, Ibaraki 305-8573, Japan
Email: yas@cs.tsukuba.ac.jp

Jinpeng Wei
School of Computing and Information Sciences
Florida International University
Miami, Florida, USA

***Abstract*—This paper describes the implementation of a parallel world model using Linux containers. A parallel world (or parallel universe) model allows a user to create multiple execution environments, called worlds, in a single operating system and to manipulate these worlds. This model enables a system administrator to create a new test world that looks like a production world. The system administrator upgrades fundamental software and tests applications in the new test world while running the production world. If the applications do not pass the tests, the administrator deletes the new world. If the applications pass the tests, the administrator merges the test world into the production world. Prior to the merge, the administrator can examine the differences between these two worlds, and identify any unintentional file modifications.**

The parallel world model has been implemented using Linux containers and Aufs, a union filesystem. In addition, this implementation uses Auditd, an audit tool, to trace file accesses in worlds. This paper describes the effectiveness of the parallel world model by analyzing a software upgrade process. Experimental results show that the world-related operations are completed within a few seconds, and that it is difficult for remote users to notice the overhead caused by running in parallel worlds.

I. Introduction

Software bugs are found every day, and software updates or patches are released regularly. System administrators, especially those in small organizations, are very busy upgrading fundamental software, such as Web servers and language processors. It is easy for system administrators to perform software updates. For example, a system administrator can perform an update by typing "`apt-get update; apt-get-upgrade`" in a Debian-based Linux distribution. However, upgrading fundamental software can crash applications. System administrators must test their applications after upgrading. If they identify problems, they must undo the upgrade.

Some systems provide transactional/undoable operations to support these activities of system administrators. Using virtual machines enables administrators to easily clone the production environment at the disk block level. However, it is not straightforward to move the contents of the test environment into the production environment after testing. Taking filesystem-level snapshots allows administrators to easily revert destructive software updates. However, this method does not help to run the production environment while testing.

To address these problems, we propose providing a parallel world model at the operating system level. A parallel world or parallel universe model is a model to describe time machines in science fiction (SF) and to interpret observations in quantum physics. In our parallel world model at the operating system level, *a world* is a container of files and an execution environment of processes. Using this parallel world model, a system administrator can upgrade fundamental software, as follows.

1) Create one or more child worlds for upgrading and testing, based on the root world as the production environment and other existing derived worlds. We allow multiple inheritance. A child world inherits files from multiple parents in a copy-on-write manner.
2) Keep running the root production world while upgrading and testing in child worlds.
3) Remove child worlds if the application does not work.
4) Merge the contents of child worlds into the root world if the application works well. Before merging, the system administrator can show differences between the child worlds and the root world, and see the files that will be modified on merging.

We have implemented this parallel world model in Linux using Linux containers (LXC) [13]. We realize the copy-on-write feature using Aufs, a union filesystem [20]. We also use Audit daemon (Auditd), a system audit tool, to trace file accesses in worlds and watch for derived files [7].

II. Commands for manipulating worlds

This section shows the usage of commands that manipulate worlds by using the following scenario.

- The production environment runs a blog application with WordPress [36], which is written in the PHP language [32]. The Web server is Apache [31].
- Patches are released for WordPress and PHP. The system administrator needs to upgrade WordPress and PHP.
- Before actually upgrading, the system administrator must test the blog application.

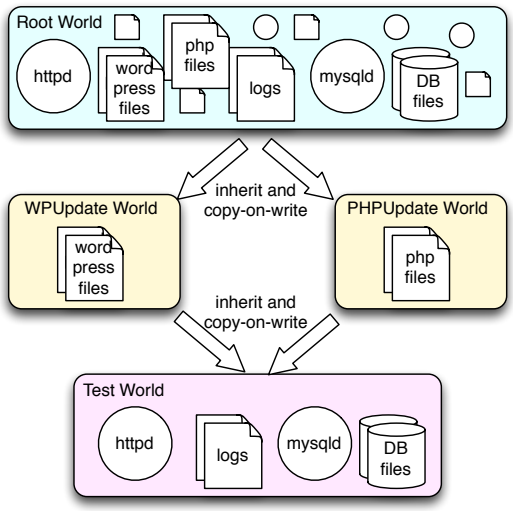


Fig. 1. A direct acyclic graph of worlds for upgrading WordPress and PHP.

- If the blog application works well in testing, the system administrator makes these patches effective in the production environment.
- Otherwise, the system administrator keeps the production environment unchanged, finds the causes of the problem and considers alternatives.

Table I shows commands for manipulating worlds. In the parallel world model, there is a special world called *the root world*. The root world, referred to as `Root`, is the base of other derived child worlds. The root world typically runs production applications. The following command runs a Web server in the root world.

```
$ wexec Root service apache2 start
```

The system administrator also uses the `wconsole` command to run commands interactively.

Next, the system administrator creates three child worlds. The first world is for upgrading WordPress; its parent is the root world.

```
$ wcreate WPUpdate Root
```

TABLE I
COMMANDS FOR MANIPULATING WORLDS.

Four fundamental commands	
Command	Description
`wcreate`	Create a child world based on one or more parent worlds.
`wdelete`	Delete a world.
`wdiff`	Show differences between two worlds.
`wmerge`	Merge a world into another world.

Supplemental commands	
Command	Description
`wexec`	Run a command in a world.
`wconsole`	Connect a terminal to a world.
`wip`	Configure IP addresses and port forwardings.
`check-off`	Checks off files on merging.

```
$ exec WPUpdate bash wordpress_update.sh
```

The second world is for upgrading PHP.

```
$ wcreate PHPUpdate Root
$ wexec PHPUpdate apt-get upgrade php5
```

The final world is for testing the blog application. Fig. 1 shows the final direct acyclic graph (DAG) of these worlds.

```
$ wcreate WPTest WPUpdate PHPUpdate
```

Next, the system administrator dumps databases in the root world.

```
$ wexec Root "mysqldump --single-transaction \
--all-databases > backup_root.sql"
```

Next, the system administrator restores databases in the test world, runs MySQL server [21], and runs the Apache Web server.

```
$ wconsole WPTest
WPTest$ service mysql start
WPTest$ mysql < backup_root.sql
WPTest$ service apache2 start
WPTest$ exit
```

Next, the system administrator maps the port of the test world to the host.

```
$ wip WPTest forward 80 50080
```

Now, the system administrator begins testing the blog application with a Web browser. If the application passes the test, the system administrator sees updated files.

```
$ wdiff WPUpdate Root
+ /var/www/html/wordpress/wp-includes/js/q
uicktags.js
+ /var/www/html/wordpress/wp-includes/sess
ion.php
+ /var/www/html/wordpress/wp-admin/plugin-
install.php
...
$ wdiff PHPUpdate Root
...
```

If no problem is found, the system administrator makes these modifications effective in the root world, and deletes the test world that contains dirty log files and databases.

```
$ wdelete WPTest
$ wmerge WPUpdate Root
$ wmerge PHPUpdate Root
$ wexec Root service apache2 restart
```

If serious problems in the patches are found, the system administrator deletes the worlds and considers alternatives.

```
$ wdelete WPTest WPUpdate PHPUpdate
```

Note that this process of upgrading and testing is not serializable in terms of transaction processing [37]. As we see in this example, we do not need strict serializability in

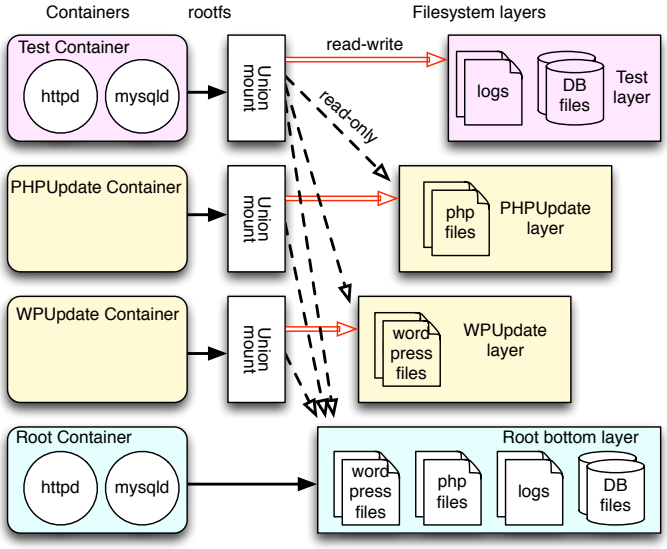


Fig. 2. Implementation of worlds with Linux containers (LXC) and Aufs, a union filesystem.

software upgrading and testing. Human administrators solve conflicts and tolerate miner problems.

In this paper, we describe the third implementation of the parallel world model. First, we implemented the model in a file server of a distributed operating system [28]. Second, we implemented it in Solaris using process tracing [10]. In these two implementations, we targeted on optimistic/speculative computation. For example, the speculative make command begins compilation before a user types "`make`."

In this paper, we describe the third implementation of the parallel world model in Linux with containers. This third implementation focuses on supporting administrators. Unlike in previous implementations, the third implementation allows running a server in the production world while running another server in a test world. Furthermore, we have added the function that collects file accesses in the root world, and help system administrators find unintentional file modifications. We will discuss about it in Section III-B.

III. Implementation of the Parallel World Model

This section describes the implementation of the parallel world model. We use containers, a union filesystem, and an audit tool.

A. Implementing worlds using Linux containers and a union filesystem

Fig. 2 shows the implementation of worlds with containers and Aufs. Each world has a container of LXC. Each container has its own root filesystem in a union filesystem [1], [9]. We chose Aufs as our union filesystem. Aufs aggregates multiple filesystem layers. The top of each stack is read-write, and the other lower layers are read-only. A process in a world can read files in lower layers. When a process writes a file, it is stored in the top read-write layer. The root world does not use Aufs.

Each world has its own network namespace. When we run a server in a child world, we can allocate an IP address to the child world, or we can map the server's port number to an unused port number of the host.

We create a child world based on a parent world as follows.

- We create a root directory with Aufs. First, we create a new empty directory. Next, we union-mount the new empty directory and the lower root directory (or directories) of the parent world(s). For example, in Fig. 2, we union-mount the WPUpdate layer and the Root bottom layer for the WPUpdate Container.
- We create a container by cloning a template container with the `lxc-clone` command. We set the root filesystem (rootfs) of the container to the directory of Aufs above.
- We configure Auditd. We will describe Auditd in Section III-B.
- We execute the container with the `lxc-start` command.

To delete a world, we kill processes in the container, delete the container, unmount the Aufs directory, and remove files in the read-write layer of Aufs.

To show differences between a parent and a child world, we show the file names in the Aufs read-write layer of the child. At this time, we exclude some system files that are automatically modified by LXC. For example, we exclude `/var/log/{wtmp,boot,dmesg,upstart}`.

To merge a source world into another destination world, we first move the files in the read-write layer of the source world to that in the destination world. Next, we migrate processes in the container of the source world into that of the destination world. Finally, we delete the source world.

To move files in world merging, we have two options.

- We retrieve the read-write layer of the source world. We also find *whiteout files* of Aufs and delete corresponding files in the read-write layer of the destination world.
- We retrieve an in-memory file access table, which will be described in Section III-B.

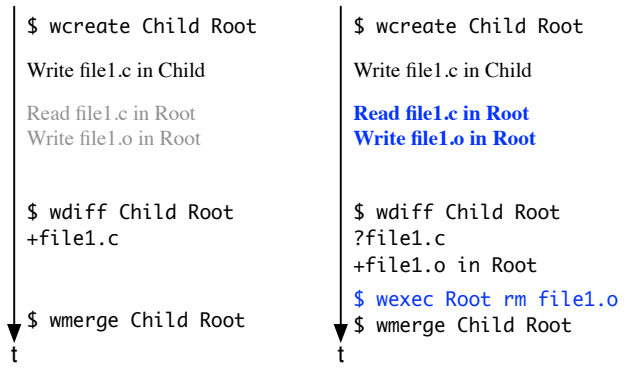
B. Collecting file accesses with Auditd

We can implement the parallel world model with LXC and Aufs. In addition, we collect file accesses in containers for the following reasons.

- Showing derived files.
- Accelerating world operations.

A derived file is a file that is generated from other files by a translator. For example, in the C language, an object file is a derived file from a source file. While Aufs can track write accesses to files, it cannot track read accesses to files. Therefore, we can miss derived files in the following case, as shown in Fig. 3-(a).

1) The root world has a source file.
2) We create a child world based on the root world.
3) We patch the source file in the child world.



(a) Without collecting file accesses in the root world.

(b) With collecting file accesses in the root world.

Fig. 3. Detecting file accesses in the root world.

4) We make an object file from the source file in the root world.
5) We merge the child world into the root world.

Although this merge updates the source file in the root world, the object file in the root world can remain unchanged. Because we keep the timestamps of files in world merging, the make command cannot detect such modifications.

To address this problem, we collect file accesses in worlds including the root world at the system call level. If we trace all the system calls including read() and write() system calls, this would cause great performance degradation. In our current implementation, we trace limited system calls as follows.

- System calls that take file names: open, rename, creat, unlink, symlink, link, utimes, utime, mknod, truncate, chmod, chown, lchown, stat, and lstat.
- System calls that take the file descriptor of a base directory: openat, renameat, and other *at system calls.

We have implemented tracing system calls by extending Auditd. The kernel code of Auditd traces system calls based on the Linux Security Module [38], and puts the results into a Unix domain socket. Our user level code reads these system call traces, and maintains a table of file accesses for each world.

We use file accesses in worlds to implement world operations, including showing differences, merging, and deleting. In the implementation of the wdiff command, we analyze the file accesses in the two given worlds. We can show warnings of conflicts to users and list the names of updated files in a parent world. In Fig. 3-(b), since the source file is written in the child world and read in the root world, the wdiff command shows a warning. In the same example, the object file is written in the root world, and the wdiff command shows the name of the object file. Therefore, the user notices and removes the object file before merging.

Since the table of file accesses includes file names in read-write layers of Aufs, we use this table to implement the wmerge and wdelete commands. Unlike in retrieving read-write layers of Aufs, we can efficiently obtain file names without disk I/O. While we do not use Aufs for the root world, we collect file accesses in the root world with Auditd. The root world can contain billions of files. We can obtain file accesses with Auditd without retrieving the entire file system of the root world while its child worlds are active.

We have modified the kernel code of Auditd as follows.

- In openat() and other system calls ending with "at," our code outputs the absolute path names of base directories.
- Our code outputs the identifiers and names of containers.

C. Process migration between worlds

When we merge a source world to another destination world, we move processes in the source world to the destination world. This allows a system administrator to use the running test world as a production world.

To realize this feature, we have added the following functions that change the attributes of arbitrary processes in a source world.

- Changing the root directory.
- Changing the current working directory and other base directories for *at system calls.
- Changing the namespaces.

It was simple to implement the first two functions by extending the chroot() and chdir() system calls, respectively. We added a process identifier (PID) argument to each system call. To realize changing the root directory of a live migrating process, for example, we perform the extended chroot() system call. This system call takes two arguments: a path name of the new root directory and a PID. We give the path name of the root directory of the destination container and the PID of the migrating process to this system call.

It was a small challenge to change namespaces, especially the PID namespace. The setns() system call is designed to set the namespace of child processes after fork(). We changed this behavior to set the namespaces of live processes. Because the current implementation changes the PID of a process after migration, an application that depends on an invariant PID will not work.

IV. Evaluation

In this section, we evaluate our implementation of the parallel world model using LXC, Aufs, and Auditd.

A. Qualitative aspect

A system administrator can upgrade fundamental software in a conventional manner using the parallel world model. The system administrator can easily create a test environment that is very similar to the production environment. In addition, the system administrator can confirm modified files before actually modifying files in the production environment.

The usage of containers in the parallel world model is different from that in Docker. Docker enables a system administrator to describe how to build the execution environment and share the description among distributed developers. Docker provides two main methods that system administrators can use to upgrade software.

- The conventional method. A system administrator runs a package management tool in containers.
- Immutable infrastructure [6]. A system administrator creates a new execution environment.

In the first method, a system administrator would want to use the operations of the parallel world mode. However, this is not the case for the second method. For example, when bugs are encountered in WordPress and PHP as in Section II, a system administrator performs the following steps.

1) The system administrator copies and modifies the Dockerfile of the production environment and creates that of a test environment.
2) The system administrator copies databases, and tests the application.
3) If the application passes the test, the system administrator stops the current production container. Next, the system administrator runs the test container as the new production container after changing configurations, again.

In the second method based on immutable infrastructure, the system administrator needs greater skills than those needed for the first conventional method. Our parallel world model supports intermediate level system administrators who use the conventional method.

In this paper, we have designed the parallel world model for system administrators, especially those in small organizations. The example in Section II has following downtime or outage duration.

- (Shutdown the Apache Web server in the Root world.)
- Merge the WPUpdate and PHPUpdate worlds to the Root world.
- (Re)start the Apache Web server in the Root world.

We can avoid this downtime of the Web server, as follows.

- Redirect incoming accesses for the production world to the test world with the wip command in Table I.
- Shutdown the Web server in the production world.
- Merge the test world to the production world.

While this procedure does not have downtime, this procedure has other problems. First, we have to reconfigure database connections if we cloned and modified databases in the test world. Second, we have to check-off log files in the test world with the check-off command in Table I if we have to keep log files in the production world. These steps can be very complex while in the example in Section II, the system administrator simply discards databases and log files in the test world by deleting the test world.

We believe that many system administrators in small organizations accept short downtime instead of performing complex steps. Such a system administrator can easily create a test environment that corresponds as closely as possible to the production environment. The system administrator can keep running the production environment while testing. The system administrator can easily remove the test environment and its contents if applications do not work in it. Finally, the system administrator can easily move the contents of the test environment into the production environment after testing.

The current implementation of the parallel world model has several limitations. First, because it depends on Aufs, we cannot effectively manage block-level file modification and directory renaming. If an application changes a single byte of a large file, Aufs copies the entire file. If an application renames a directory, Aufs also performs copying recursively.

We need an improved union file system that effectively manages these problems [39]. Linux 4.5 introduces a new system, call copy_file_range() and its VFS layer API in struct file_operations [14], [15]. This API allows a file system to share data blocks between a source file and destination file in copying. Btrfs [25] makes use of this new feature in Linux 4.5. In our future work, we would like to use this feature to implement an efficient union file system for the parallel world model. In addition, we can omit collecting file accesses with Auditd if a file system does it with container identifiers.

In the current implementation, we cannot manage index files of package management tools. For example, the Debian package manager (dpkg) uses the file `/var/lib/dpkg/status`. If we create two parallel worlds, and both worlds update this file by installing some packages, one of two updates will be lost. To avoid this, we need to run the package manager in a single world, or we need to create a child world of a child world. We need an improved package manager that does not use such centralized files. We can implement a new package manager that makes maximum use of the parallel world model, in a similar manner as the speculative make command [10], [28].

B. Performance

We performed experiments to demonstrate the usefulness of the implementation of the parallel world model with containers. We measured the following execution times.

- Execution times of four fundamental commands. We used the command interaction in Section II.
- Access times and throughputs of WordPress running in a world. We used Apache Bench (ab) to measure access times and throughputs.

Table II shows our experimental environment. These two PCs were connected with 1 Gbps Ethernet. We ran Apache httpd 2.4.7, PHP 5.5.9, WordPress 4.0, MySQL 5.5.46, and Apache Bench 2.3. In the second experiment, we disabled caching for WordPress. We posted 1000 articles to WordPress. The size of each article was 6500 bytes. We configured the ab command to access the top page of the blog application.

TABLE II
EXPERIMENTAL ENVIRONMENT.

	WordPress PC	Apache Bench PC
CPU	Intel Core i7 870 2.93GHz 4 cores, Hyper Threading	Intel Core i7-4790K 4.00GHz 4 cores, Hyper Threading
Memory	32 GB	32 GB
Kernel	Linux 3.13.11 modified	Linux-3.13.0-36-generic
Distribution	Ubuntu-14.04 (64 bit)	Ubuntu-14.04 (64 bit)

The size of the page was 76 KB. We set the number of simultaneous connections to 10 and the number of total requests to 100.

Table III shows execution times of commands for manipulating worlds using the in-memory file access table described in Section III-B. We also performed same experiments by retrieving the read-write layers of Aufs. However, we have obtained no significant difference between them because all Aufs data was cached in the main memory.

Creating a world took a relatively long time because we created and started an LXC container. The execution times of wmerge and wdiff depended on the number of files. Table IV shows the numbers of files and their total size for each child world. When the number of files was a few hundred, the execution times for merging were less than two seconds. When we ran the wdiff command in a terminal, the display speed was bounded by the terminal. The wdelete command ran fast because we did lazy execution of cleaning up files in the Aufs read-write layer and destructing the container.

Table V shows the access times in seconds and throughputs in request per second when we accessed the top page of WordPress with a warm filesystem cache. We measured performance with and without Aufs and Auditd. When we ran without Aufs and with Auditd, we obtained the performance of the root world. When we ran with both Aufs and Auditd, we obtained the performance of a child world. As shown in this table, the overhead incurred by the worlds was less than 12 %. If a user accesses the blog application over the Internet, it is unlikely that they would notice this overhead. We believe that most system administrators will accept this small performance degradation, especially while they are testing.

V. Related work

We discussed comparisons with Docker in Section IV-A. Our parallel world model supports intermediate-level system administrators, while Docker supports senior-level system administrators.

Researchers have implemented transactional/undoable operations at the system call level [4], [11], [17], [19], [22], [23], [26], [27], [34]. Between these systems, the system in [27] most closely resembles our parallel world model. Using Sun ZFS in Mac OS X, this system allows a system administrator to upgrade software while retaining the existing version. Some version control systems can be accessed through filesystem interfaces [3], [5], [12], [18], [24]. An advantage of our parallel world model over these systems is that we support concurrent execution of both the production environment and a test environment.

Using virtual machines enables administrators to easily clone the production environment [29], [33], [35]. However, it is not straightforward to move contents of the test environment into the production environment after testing.

Distributed shared repositories use various types of merging semantics and conflict resolution methods [8], [16], [30]. In our parallel world model, we use simple merging semantics for a single system administrator. Our model allows a system administrator to check off insignificant log files before merging.

VI. Conclusion

In this paper, we have described the implementation of a parallel world model using Linux containers. This model enables a system administrator to create a new test world that looks like a production world. The system administrator can upgrade fundamental software and test applications in the new test world while running the production world.

We have implemented the parallel world model using Linux containers, Aufs, and Auditd. We have shown the effectiveness of the parallel world model by analyzing a software upgrade process. Experimental results show that the world-related operations are completed within a few seconds, and that it was difficult for remote users to notice the overhead caused by running in parallel worlds.

In our future work, we plan to implement an efficient union file system for the parallel world model. We are also interested in making the parallel world model work together with other container technologies, including Docker and Rkt [2]. We would also like to implement an efficient union file system for the parallel world model.

TABLE III
Execution times of world operations.

Operations		Execution times in seconds
wcreate	WPTest/WPUpdate Root	0.85
wcreate	WPTest WPUpdate PHPUpdate	1.10
wmerge	WPUpdate Root	1.15
wmerge	PHPUpdate Root	1.20
wdiff	WPUpdate Root	0.04
wdiff	PHPUpdate Root	0.05
wdelete	WPTest	0.03

TABLE IV
Numbers of files and their total sizes.

Worlds	# of files	Total sizes in MB
WPUpdate	45	1.9
PHPUpdate	219	87.5
WPTest	30	1.9

TABLE V
Average access times and throughputs of WordPress.

Aufs	Auditd	Access time in milliseconds	Requests per second
-	-	231	41.5
-	✓	250	38.5
✓	-	231	41.6
✓	✓	258	37.4

REFERENCES

[1] N. Brown. Overlay Filesystem. Accessed: 2016-02-01. [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt

[2] CoreOS. rkt, a security-minded, standards-based container engine. Accessed: 2016-02-01. [Online]. Available: https://coreos.com/rkt/

[3] B. Cornell, P. A. Dinda, and F. E. Bustamante, "Wayback: A user-level versioning file system for Linux," in *Proceedings of Usenix Annual Technical Conference, FREENIX Track*, 2004, pp. 19–28.

[4] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "ReVirt: Enabling intrusion analysis through virtual-machine logging and replay," in *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI)*, 2002, pp. 211–224.

[5] J. Epler, D. Patterson, and V. Rutsky. FUSE-powered filesystem over Subversion repository. Accessed: 2016-02-01. [Online]. Available: https://github.com/rutsky/svnfs

[6] C. Fowler. (2013, Jun.) Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components. Accessed: 2016-02-01. [Online]. Available: http://chadfowler.com/blog/2013/06/23/immutable-deployments/

[7] S. Grubb. Linux Audit Project. Accessed: 2016-02-01. [Online]. Available: http://people.redhat.com/sgrubb/audit/

[8] R. G. Guy, J. S. Heidemann, W.-K. Mak, T. W. Page Jr, G. J. Popek, D. Rothmeier *et al.*, "Implementation of the ficus replicated file system." in *USENIX Summer*, 1990, pp. 63–72.

[9] D. Hendricks, "The translucent file service," in *Proceedings of the Autumn 1988 EUUG Conference*, 1988, pp. 87–93.

[10] K. Ishii, Y. Shinjo, and K. Itano, "The implementation of the World OS using a process trace facility," *Journal of Information Processing Society of Japan (IPSJ)*, vol. 43, no. 6, pp. 1702–1714, 2002.

[11] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005, pp. 1–1.

[12] D. G. Korn and E. Krell, "The 3-D file system," in *Proceedings of the USENIX Summer Conference*, 1989, pp. 147–156.

[13] Linux Containers Project. Accessed: 2016-02-01. [Online]. Available: http://linuxcontainers.org

[14] Linux Kernel Newbies. (2016, Mar.) Linux 4.5. Accessed: 2016-03-16. [Online]. Available: http://kernelnewbies.org/Linux_4.5

[15] LWN.net. (2015, Sep.) copy_file_range(). Accessed: 2016-03-16. [Online]. Available: https://lwn.net/Articles/659523/

[16] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières, "Replication, history, and grafting in the Ori file system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, 2013, pp. 151–166.

[17] C. B. Morrey III and D. Grunwald, "Peabody: The time travelling disk," in *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2003, pp. 241–253.

[18] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok, "A versatile and user-oriented versioning file system." in *USENIX Conference on File and Storage Technologies*, vol. 4, 2004, pp. 115–128.

[19] E. B. Nightingale, P. M. Chen, and J. Flinn, "Speculative execution in a distributed file system," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*, 2005, pp. 191–205.

[20] J. Okajima. Advanced Multi Layered Unification Filesystem. Accessed: 2016-02-01. [Online]. Available: http://aufs.sourceforge.net/

[21] Oracle Corporation. MySQL Developer Zone. Accessed: 2016-02-01. [Online]. Available: http://dev.mysql.com/

[22] Z. Peterson and R. Burns, "Ext3cow: A time-shifting file system for regulatory compliance," *ACM Transactions on Storage*, vol. 1, no. 2, pp. 190–212, May 2005.

[23] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel, "Operating system transactions," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, 2009, pp. 161–176.

[24] Presslabs. PressLabs/gitfs: Version controlled file system. Accessed: 2016-02-01. [Online]. Available: https://github.com/presslabs/gitfs

[25] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-Tree filesystem," *Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 9:1–9:32, Aug. 2013.

[26] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir, "Deciding when to forget in the Elephant file system," in *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*, 1999, pp. 110–123.

[27] D. Spinellis, "User-level operating system transactions," *Software: Practice and Experience*, vol. 39, no. 14, pp. 1215–1233, Sep. 2009.

[28] J. Sun, Y. Shinjo, and K. Itano, "The implementation of a distributed file system supporting the parallel world model," in *Proceedings of The Third International Workshop on Advanced Parallel Processing Technologies*, 1999, pp. 43–47.

[29] P. Ta-Shma, G. Laden, M. Ben-Yehuda, and M. Factor, "Virtual machine time travel using continuous data protection and checkpointing," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 1, pp. 127–134, Jan. 2008.

[30] V. Tao, M. Shapiro, and V. Rancurel, "Merging semantics for conflict updates in geo-distributed file systems," in *Proceedings of the 8th ACM International Systems and Storage Conference*, 2015, pp. 10:1–10:12.

[31] The Apache Software Foundation. The Apache HTTP Server Project. Accessed: 2016-02-01. [Online]. Available: https://httpd.apache.org/

[32] The PHP Group. PHP: Hypertext Preprocessor. Accessed: 2016-02-01. [Online]. Available: http://php.net/

[33] VMware. Understanding Clones. Accessed: 2016-02-01. [Online]. Available: https://www.vmware.com/support/ws5/doc/ws_clone_overview.html

[34] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS distributed operating system," in *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 1983, pp. 49–70.

[35] J. Wires and M. J. Feeley, "Secure file system versioning at the block level," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '07)*, 2007, pp. 203–215.

[36] WordPress.org. Blog Tool, Publishing Platform, and CMS — WordPress. Accessed: 2016-02-01. [Online]. Available: https://wordpress.org/

[37] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok, "Extending ACID semantics to the file system," *ACM Transactions on Storage (TOS)*, vol. 3, no. 2, Jun. 2007.

[38] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: General security support for the Linux kernel," in *Foundations of Intrusion Tolerant Systems*. IEEE, 2003.

[39] X. Wu, W. Wang, and S. Jiang, "Totalcow: Unleash the power of copy-on-write for thin-provisioned containers," in *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys '15)*, 2015, pp. 1–7.