# OSXerox: Live Copy of a System into a VMM Running on Itself

Erik Dietrich and David McCloskey
*University of Illinois at Urbana-Champaign*
{*ediet2,mcclosk1*}*@illinois.edu*

## Abstract

Average computer users frequently compromise their own systems by visiting web sites and installing software. Meanwhile, industry frequently makes use of virtualized copies of running operating systems which can be migrated and discarded at will.

This paper describes *OSXerox*, software which can create a virtualized copy of a running operating system, without shutting down, and run it on the same system. OSXerox creates a live copy of a running operating system by duplicating both the filesystem and the list of running processes of the source operating system into a virtual machine monitor running on the same machine.

We have implemented OSXerox using the Tiny Core Linux distribution and QEMU. Our results show that the time to create and boot the virtualized copy of a Tiny Core Linux system increases linearly with the size of the source filesystem and is dominated by the time required to boot the system. In order to begin working with the virtual copy of their machine, users can be expected to wait for about the same amount of time as it takes the system to boot.

## 1 Introduction

Virtual machine technology has become ubiquitous in today's computing world. Many developers and savvy users make explicit use of virtual environments for experimentation or compatibility. Casual users indirectly use them by virtue of the fact that sites they visit or tools they use may be hosted in virtual environments [10]. These environments offer safety through sandboxing, efficiency through better use of hardware, and flexibility through portable environments. This flexibility is one of the reasons that virtualization is so popular; users can run applications that were written for specific computer systems regardless of their current computer's operating system or processor architecture [2].

While there are methods for migrating virtual OS from one host to another [4, 9] and migrating between virtual and physical OS [25], no methods exist for creating a virtual copy of a physical OS on the same machine. The ability to do so could both enhance the user experience and have important security benefits. Users would be able to try out new applications, OS settings, browser plugins, and device drivers without worrying about negative effects or hassling with cumbersome system restores [22]. System administrators could automatically "go virtual" when clients make requests with security implications. Honeypot creators could allow malware to execute on bare hardware and then go virtual to study the effects and ramifications of the run [1].

The use of both computing in general and virtual machines specifically is only going to grow in the coming years, increasing the demand for OS portability [7]. Anything that adds to the fluidity of virtual environments has the potential to enhance the user experience and the security, availability, and reliability of providing software services.

No solution for creating a virtual clone of a running physical OS on that same physical OS exists. Various vendors provide tools that can migrate existing physical OS installations to the vendors' own VMMs [26, 27]. However, these require that the machines be shut down and modified before migration. Researchers have developed methods for live migration of virtual machines between VMMs [4, 9], but these have only been implemented for systems running on a hypervisor, not bare hardware. VMWare provides a proprietary *hot cloning* tool which migrates an OS from a physical machine to a VMM running on a different machine [25], but this requires the use of two machines.

In general, the aim of existing live VM migration approaches is to facilitate load balancing across physical servers or to create copies that aid in maintenance [4]. These approaches [25] require two physical machines: the existing machine and a new machine to run the

VMM. Our solution allows users the benefits of virtualizing with only one machine and is a tool for general use rather than an add-on to an existing commercial offering.

We present OSXerox which runs on a plain, installation of Linux and uses arbitrary virtualization software to create a copy of the physical machine OS. It does not require an underlying hypervisor, two separate machines, a reboot, or any custom made software. It works with the OS and the virtualization software to create a replica of the running physical host as the physical host continues to run.

OSXerox accomplishes this goal by means of creating and customizing an ISO image that can be used by the VMM software to boot a guest. Our design and implementation make use, specifically, of the *Tiny Core Linux* [24] distribution and QEMU [18] as a VMM. However, with modification only to meta-data and OSXerox modules, this can be expanded to other distributions and VMMs.

Modularity and extensibility are at the core of OSXerox's design and usefulness. By separating the processes of analyzing the physical machine, extracting and modifying an ISO, and launching the virtual machine, it creates an actual framework of OS replication and can be used, modified, and expanded to allow an intuitive and flexible replication of a user's OS.

With Tiny Core and QEMU, we have proven our design by deploying deliverable code to our test environment and executing it. Our results show that the user can, with properly configured meta-data, execute a simple command line invocation that builds and launches a VM containing a reasonable facsimile of his physical environment. The cloned environment is not identical; kernel state and memory values are not guaranteed to be exactly the same, but they will be similar. The filesystem and running processes, with the exception of the OSXerox process to avoid recursion, are identical.

# 2 Design

## 2.1 Design Goals

Our primary design goal was to create a virtualized copy of a running system within the same without shutting down and restarting. Toward that end, we also enumerated three sub-goals: maximizing the fidelity of the copy, minimizing the downtime experienced by the user, and keeping the system as flexible and portable as possible.

We worked on two approaches for achieving these goals. For one, a full copy of all memory pages in both kernel and user space would be copied into the VM. This is a high-fidelity copy whose performance would vary linearly with the size of the memory needed for copy. The other approach would be a lower-fidelity approach that involved creating an ISO image on the fly and booting the VM using it. Its performance varied in the same way but with additional overhead, but its advantage was in conferring a greater degree of flexibility by not requiring modifications to the VMM or the kernel of the target system. Ultimately, we chose the medium fidelity approach, but significant work was done along both paths and both designs are detailed in this section. The full memory duplication approach is also worth including because OSXerox creates a foundation upon which full memory duplication can be added as a natural extension.

## 2.2 Full Memory Duplication Approach

We modify QEMU to accept a stream containing memory pages and use them to start an already running system. We decided to pursue this approach as it allows us to modify virtual processor registers (like the stack and frame pointers and the location of the page tables) before the copied system begins running. Other approaches we considered include a bare-bones operating system which would receive memory pages over the network and using an existing boot loader to boot a kernel image which would run and receive the memory pages.

With the QEMU process ready to receive memory pages, we invoke a kernel module which stops all the other machine operations and copies all memory pages to the virtual machine (Figure 1). Migration of memory can take several forms – pre-copy, stop-and-copy, and post-copy. There must be some combination that at least contains stop-and-copy. In general, these methods involve an iterative copying scheme where pages are marked as dirty when they are otherwise accessed. Once some small number of pages has been modified in an iteration of copying, the system is stopped and the remaining pages are copied. In our system, we believe that a pure stop-and-copy system is fast enough because all memory accesses take place within the machine.

We use a kernel module to access the physical memory pages of the running machine directly. The `mem_map` array contains the array of all physical pages which can be mapped and unmapped temporarily into kernel memory using the `kmap()` and `kunmap()` functions. We map each page individually and then copy it into the QEMU memory. Before we choose to copy a page, we verify that it is not owned by the QEMU process.

Once memory pages have been received by the virtual machine, several changes must be made. Due to the change in physical memory size, several parts of kernel state must be changed. There are several variables which depend on the size of physical memory and are updated. Additionally, the physical location of memory pages will change, and the pages tables for the kernel and all processes must be updated.

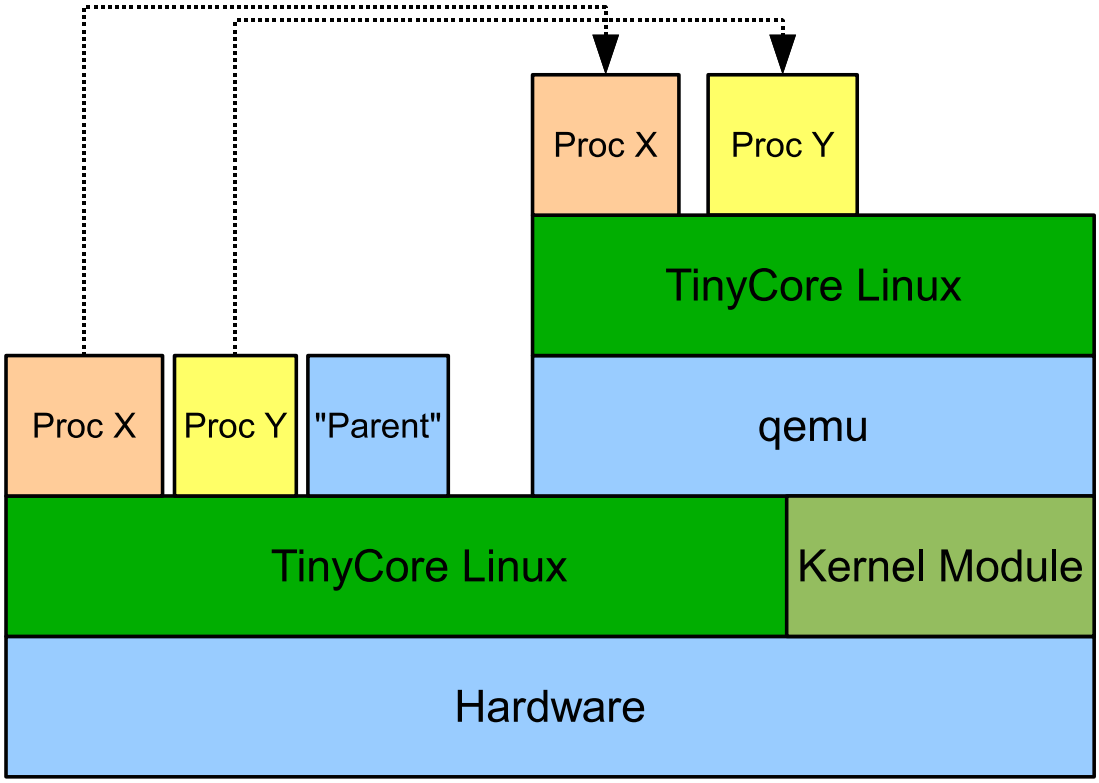


Figure 1: A *parent* process launches the VMM and inserts a kernel module. The kernel module copies all allocated physical memory into the VMM, producing a replica of the running operating system.

We explored an alternative design where we copy all memory of individual processes from a user-space program. Using this method we would access each process and use the `/proc/pid/mem` psuedo-file to access all of process memory. From here, we would copy this memory into newly created processes within the virtual machine. By simply avoiding the QEMU process as a source when copying, we prevent recursion.

When using the user-space copying method, we have two alternatives for copying the kernel to the virtual machine. The first is to use the kernel module to copy itself directly into the virtual machine. This approach may be troublesome because the kernel data structures will be invalid and will need to be modified before starting the virtual machine. The second approach is to use the existing kernel image to boot a new kernel within the virtual machine.

### 2.2.1 Copying Memory

Several methods for copying memory have been examined in the frame of virtual machine migration and process migration. The drawbacks of these methods have been considered with a focus on minimizing downtime and total migration time between different physical hosts across local or wide area networks. Because we aim to migrate to the same physical host, we do not have to be as concerned with the performance with respect to time, but optimizations could be made which reduce total memory usage.

The process for copying memory can be separated into three phases with respect to when the target machine becomes operable:

**Pre-copying** The source machine continues to run while copies of memory pages are made to the target machine. If any pages are modified during this phase, they need to be re-sent to the target.

**Stop and copy** The source machine is stopped and memory pages are copied to the target. Then, both are started.

**Post-copy** The target machine is running and any missing pages are demand-copied from the source.

Several approaches have been taken with respect to using one or a combination of several of these processes.

*Pre-copy* [4] involves a bounded number of iterations of copying pages before the source machine is stopped and any remaining, modified pages are copied to the destination. The pages that remain to be copied are identified as a *writable working set* which correspond to those processes that are actively writing to memory on an ongoing basis.

*Pure demand-migration* [28] uses a process whereby a short stop-and-copy phase is followed by on-demand post-copy operation of the target virtual machine. This is not the approach we take in our design, but it certainly may be an optimization available for future work. By modifying the hypervisor to service page faults to the original host OS combined with a copy-on-write scheme to preserve the original state of all memory pages, the total memory usage could be reduced considerably.

The approach taken in our design is *pure stop-and-copy* [19, 13] whereby all activity on the source machine is halted and the contents of memory are copied to the target machine. The primary concern with taking this approach is that the downtime the user experiences may be too great. However, our initial tests have shown that the time to copy memory within a single physical computer system is small enough that an interactive user should find the downtime acceptable. Because we are creating a *copy* of a running machine at the request of an interactive user, we assume that there are no time-critical services running on the machine that would not tolerate the downtime.

### 2.2.2 System Migration

Past work has primarily been divided into two approaches for migration of operating systems: host-driven migration and self-migration.

In *Host-driven migration*, a system is migrated by the system it is running on. Typically this applies to either process migration or migration of a running virtual machine. The operating system (in the case of process migration) or the hypervisor (in the case of virtual machine migration) drives the migration by handling packaging, memory replication, and resource migration.

The approach taken, and required, by our design is *Self-migration* [8, 4]. This is the process by which an operating system migrates itself to a new platform. This approach is required because the source machine is running directly on hardware, so without hardware support, there is no ”host” to migrate the operating system. One drawback of this approach is that it needs to be implemented separately on each type of operating system. Our design is only implemented on the Linux 2.6 kernel, and we have not examined its application to other operating systems.

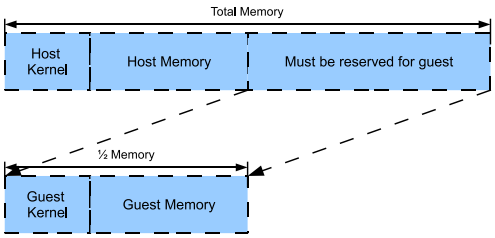


Figure 2: Because we wish to copy all in-use memory to a guest running on the same machine and we do not have swap space, we are limited to less than half of physical memory.

### 2.2.3 Limitations

The exact duplication of the virtual machine is limited in different ways by the alternatives of our design. If an exact copy of all physical pages is made, the virtual machine will be an exact copy of the physical machine. If the kernel itself is copied but individual process memory is copied separately, the kernel will not be an exact copy. For example, the process identifiers may be different, and the page tables will be different due to the time each page was allocated. If we use a kernel image to boot the virtual machine, and then copy the processes, the differences from the previous example exist as well as new differences derived from not having copied the kernel state.

All of our designs are limited by the size of memory (Figure 2). Because we seek to make an exact replica of the running machine, the available physical memory must be identical. If swap space is not available, we must limit the data copied to the virtual machine to be less than half of the memory of the physical machine. If a swap space were available, it would be possible to allocate a full memory space to the virtual machine, but the normal concerns with thrashing would exist. Additionally, the performance of the copying process itself would be impacted if some pages had to be swapped to disk.

Our design assumes that there are no mounted disks or open network connections. It is not possible to make a direct copy of a machine that has mounted a disk or opened a network connection in a read/write manner and have both copies running with the assumption that they have exclusive access to the devices. Future work could be performed on multiplexing network connections or making a copy-on-write virtualization of a disk drive that becomes part of the virtual machine’s state.

## 2.3 ISO Generation Approach

The second angle of approach and the one that we ultimately used was creating a disk image (ISO) to use for booting the guest system. This approach is simpler than the memory copy approach. We reason that one way to generate a customizable guest OS is to produce an ISO that contains our customizations.

Our design addresses two major challenges: generating an ISO based on the host OS and communicating with the newly created guest. Generation of the ISO is addressed, specifically in Tiny Core, by relying on the assumption that the OS is always booted from a fresh ISO. Since Tiny Core is a memory-only operating system, this is a reasonable assumption. In general, we accomplish this by requiring as part of our deployed package that the user provide access to the original OS ISO. We communicate with the guest via files that we place in its ISO. All communication in our base facility is thus asynchronous and involves a one-time, strategic placement of files within the target filesystem.

At a high level, the following tasks are performed in order to generate the guest OS.

**Reconnaissance** We locate the base ISO image and make it available.

**Duplication** The base ISO is modified to contain the contents of the host filesystem.

**Asynchronous communication** We communicate with the guest by strategically adding the files needed to manipulate the its state into mimicking the host's state.

**Launching the Guest** We launch QEMU using the modified disk image.

The details of each of these tasks are discussed in the following sections.

### 2.3.1 Reconnaissance

Upon launching OSXerox, we first obtain a base ISO image to work with. In Tiny Core, this is not a particular problem, since it has necessarily been booted from a mountable ISO in order to run. This might come from a CD, USB thumb drive or network boot. To allow for this flexibility, our tool can be configured with the mount location as a piece of meta-data.

Once the ISO is mounted, we copy its contents to a temporary local directory so that we can modify it and unmount the ISO. This is all done in the interest of conserving memory since file copying consumes memory in a memory-only OS. Once the files are copied, we take the zipped filesystem out and extract it.

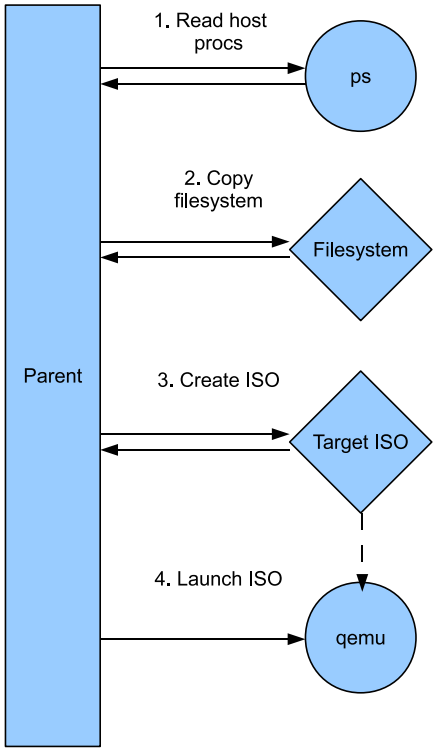


Figure 3: OSXerox reads the host process list (reconnaissance), duplicates the filesystem, creates the disk image with control files used for communication with the guest, and then launches the quest in QEMU.

### 2.3.2 Duplication

Next we modify the extracted base filesystem to create a duplicate of the host. The idea is to create a new ISO that contains the filesystem of the host. However, this is not a simple matter of copying the contents of the filesystem root to the target. There are folders such as "proc" and "sys" that need to be created in the booting guest. So, we have abstracted the folders to be copied into a meta-data file. This file allows a per-installation configuration of how the target filesystem is generated.

This file contains two entries: one with an absolute path on the host and the second with a relative path in the target filesystem. Our utility reads these entries and builds the target accordingly. This strategy allows a great deal of flexibility in configuring the target filesystem and thus the installed programs and settings of the guest.

#### 2.3.3 Asynchronous Communication with Guest

Once filesystem duplication is complete, we create files which will provide instructions to the guest once it begins execution. It is not sufficient to launch the guest with merely the same filesystem in place. This would create a guest with the same installed programs and OS configuration, but not the running processes and non-persistent user data.

To provide this information, we make use of the way a booting OS configures itself from its files. Specifically, we modify the booting and loading files to put the guest in the state that we want it. In order to ensure that the processes we want are running in the guest, we gather their command line invocations and put them in a shell script that will be executed once the guest's window manager is loaded and the login is complete.

On Tiny Core, this is not trivial. On boot, Tiny Core loads up a pre-configured .xsession for the base user and deploys it. We cannot simply put our command line invocations into the guest's home directory .xsession; we must put it into the pre-configured file that will be loaded. On other Linux distributions, this would not be necessary; we could simply make use of a file like /etc/rc.d/rc.local. However, we need to account for the fact that this particular OS has no persistence and thus configures a canned set of user data on each boot. Another advantage of the meta-data configuration of deploying files into the target filesystem is that it allows this to be configured per OS, post deployment.

#### 2.3.4 Launching QEMU

Once we have our target filesystem, including the files used to communicate with the guest, it is time to create the actual ISO that we will use. This is done by repacking the target filesystem, and swapping it into the ISO file structure that we created during step 1. With this, we use mkisofs [15] to create the final ISO file. From here, we launch a QEMU guest with the ISO and allocate most of the system's remaining memory to running the guest (leaving a bit for errors, and erroring out if insufficient memory is available).

#### 2.3.5 ISO Approach Advantages

One of the main advantages of this approach is that it is highly agile. The meta-data file that governs the process makes the system quite robust in the face of any future changes. Updates to the user's OS or a change in the VMM software being used can be addressed by simply tweaking meta-data settings. For most such changes, no code modifications would be needed. And, in the event that code modifications would be needed, the modular design allows for changes to be isolated within a single executable module and thus obviating once again the need for a full re-deploy. This modularity also allows savvier users to plug in their own methodology for one section of the application.

Another advantage of this system is that it is highly extensible. ISO is fairly standard and our customizable scheme allows for flexibility in generating these ISOs. This allows our system to be used in arbitrary Linux distributions and probably even other OS such as Windows.

A third advantage of using the ISO generation approach is that there are lots of opportunities for streamlining. Easy add-ons or modifications would include pre-generation of a filesystem copy using a daemon, rapid memory copy to guest by deploying a "listener" within the guest kernel, a built-in core ISO image, etc. All of these additions allow for mitigation of the performance loss we suffer with this approach versus memory copy.

A final important advantage is that the ISO approach allows for the eventual inclusion of the memory copy approach. Future work on the project could include generating an ISO that boots a custom kernel which, in turn, performs memory copy to the guest and then hands over execution. By choosing the implementation method that we did, we are able to build on and refine our results.

## 3 Implementation

The implementation of OSXerox is highly modular. This was done both to confer the aforementioned advantages on the implementation and to make implementation easier and more configurable. One of the development difficulties in our approach is that troubleshooting the launch of a virtual machine is time consuming since it is necessary to wait for a system to boot and to observe its state after each change is made. As such, implementation naturally tended toward modularity and a dutiful avoidance of repeating any steps unnecessarily.

OSXerox consists of a parent process called "clone" and two modules, implemented in C++, and some meta-data. In a deployed system, these modules and the parent all reside in the same directory along with the meta-data files. In the future, there might be some motivation to introduce a hierarchical directory structure, but with as few files as there are currently, doing so did not make sense. The parent module invokes these sub-modules, which are called "getprocs" and "makeiso". Via command line parameters, the parent and sub-modules all share meta-data as needed.

### 3.1 Clone

The parent module, "clone", handles processing from a conceptual, high level standpoint. The first thing it does

is delegate to ”getprocs”, which handles returning all host process data and stores it for processing and addition to the target ISO. The exact mechanism for this will be addressed later in the section on meta-data.

Once the host processes are recorded, the parent creates a temporary directory in the root of the filesystem. It copies the ”makeiso” module into that directory, along with the meta-data that the process requires. This is needed to avoid recursive filesystem copying during the generation of the target ISO. From there, it invokes the ”makeiso” module to create the target ISO. It copies the target ISO back to the working directory and then removes the temporary working directory from the filesystem root.

With the ISO in place, the parent process invokes QEMU and waits for the user to be finished with the virtual machine. From there, it exits, and cleans up the target ISO. Once it is finished, the host system is left in the same state that it was in prior to the user invoking the clone command.

## 3.2 Getprocs

The ”getprocs” utility is responsible for collecting process information from the host. This is implemented by making use of the output of the ”ps” command line utility and performing several manipulations of the data. First, ”ps” is queried to obtain the pid, command line invocation and command line arguments of the processes, filtered by processes belonging to the logged in user. With this information, a series of system calls to stat check the proc filesystem to see if the process is running. This is useful for filtering out ephemeral processes such as the calls to ”grep” and ”clone” that turn up as a result of OSXerox and which should not be duplicated in the guest.

For the remaining processes, we obtain their root directory of execution from the proc filesystem. At this point, we have all of the information that we need for duplication. We add the command line invocation with command line arguments to the target xsession file and bracket it with a change directory to the process root directory, and then a ”cd -” back to the original directory. In this fashion, we recreate all of the relevant user commands on login. It is important to note that using the session file rather than the .profile or user rc file is imperative because if the user has a terminal open, these latter two will result in recursive terminal launching.

## 3.3 Makeiso

The functionality of ”makeiso” is covered in detail in the design section. Indeed, makeiso is the core of our design and implementation in that it is responsible for actually constructing the target image. We have previously mentioned our motivation for the modular approach, so it is worth mentioning here why we did not feel it prudent to divide the function of makeiso into smaller, more modular components, since it covers so much ground. While the functionality of extracting, modifying, and packing an ISO is an involved procedure, the actual modification of an ISO to suit our purposes is atomic from a conceptual standpoint. In order to introduce as much flexibility as possible, however we did abstract out much of the filesystem modification into the meta-data file ”files.txt”

## 3.4 Meta-Data

The backbone of the meta-data is ”files.txt”. This file contains a series of tuples used during ISO generation. These tuples take the form of (source, target) where ”source” represents an absolute path on the host system and target is a relative path inside of the ISO being built. In this fashion, the user is able to specify individual files or entire directories which may be recursively copied. In addition, users can change filenames and locations, which allows our implementation to arbitrarily generate ephemeral files and insert them on the fly into the target. This is invaluable for implementing the asynchronous communication with the guest.

In fact, this is exactly how the ”getprocs” module is able to convey process information to the guest. It appends process data to another meta-data file, ”xsessions”. xsessions is, in Linux systems, the file that invokes various commands per user login. Generating a custom xsessions file through meta-data allows cross-distribution flexibility and also an easy method of communicating with the guest: we write processes we want invoked into a file that the guest reads for commands to invoke.

A final piece of meta-data is called ”clone.cfg”. In this file we store information about the source of the ISO (i.e. a CD-ROM, USB boot stick, hard drive, etc), the name of the temporary directory, the name of the guest ISO to be generated, and various other information that assists with configuration and need not be hard-coded. The idea here is to allow as much runtime flexibility as possible. A system administrator or a savvy user could change the behavior of OSXerox to suit his needs or a deployment package could be created along with a wizard that allows for install-time generation of settings.

# 4 Evaluation

Our main goal with this effort was to create a very specific live physical to virtual migration: a physical migration to a virtual machine running on the physical system.

Our secondary goals were to achieve as high a level of fidelity as possible in doing so and to minimize the downtime experienced by the user. In the subsequent sections, we will explore in depth our results as compared to our goals.

## 4.1 Live self-migration

OSXerox successfully performs the live self-migration of an operating system. From our source code and meta-data, we created a tar ball for deployment purposes. We then booted up a plain Tiny Core OS running on a computer with 1 Gigabyte of memory and a 2.66 GHz, single core processor. From there, since Tiny Core is a memory-only OS [24], we installed the bare essentials needed in order to run the OSXerox utility: QEMU and mkisofs-tools. We configured meta-data to reflect the system on which we were running: the original boot medium was a CD ROM and the filesystem paths targeted for copy were all files save system generated ones such as the proc filesystem and the /sys directory.

With the setup in place, the unzipped and executed OSXerox process and observed that a QEMU guest was launched and booted with the same user processes running inside of it as were running on the guest. Processes that were successfully launched in subsequent trials with more software installed on the host included direct command line invocations such as text editors, GUI initiated applications such as thy Tiny Core app browser, and GUI applications run from the command line such as the text editor "scite" [20].

In addition to verifying running processes, we also verified information about the filesystem structure. This was done mainly through spot-checking and note-taking since networking with the guest is both cumbersome and beyond the scope of our initial effort. To streamline our spot checking, we selected directories that were known to see the most frequent traffic in the course of development and testing such as the user home directories, the Tiny Core applications directories and the settings directories in /etc. In all cases that we checked, these files were identical. The only non-identical files in the guest were the ones that we did not include in the file meta-data for copy and ones that we specifically made different from the host in order to allow asynchronous communication.

## 4.2 Fidelity

Our design creates a VM copy with a high degree of fidelity in recreation of the filesystem and a reasonable degree of fidelity in recreation of running processes from a user standpoint. From the user perspective, this creates a reasonable facsimile and the illusion that the two systems are nearly identical.

However, the reproduction is not as faithful as it could be. Our implementation treats the OS and kernel state as a black box. Because all communication and reproduction is done in user space, we have no reliable metric for evaluating the degree of successful copy of OS internals and thus, we cannot assume any real degree of accuracy. The two kernels may, in fact, be quite different from one another.

In addition, we make no serious attempt to reproduce the host's user space memory state in the guest. There will be some overlap at a macroscopic level, since we are reproducing the host's running processes. However, this rather weak assumption of similarity is the best that we can offer. Because of our lack of visibility into the kernel, we do not have access to information such as page tables and memory state in the guest or the copy. Furthermore, while we reproduce processes in the guest by re-creating their command line invocations, we are not able to modify process memory beyond that at this time. For example, if a user opened an empty text file for editing and added the word "test" to it without persisting the changes to disk, a subsequent run of OSXerox would result in an instance of the text editor with an empty file. Because the word "test" exists only in the host process's memory, it will not appear inside of the OSXerox guest.

The logical ramification here is that prospective users, particularly those interested in the sandboxing and experimental uses of OSXerox, must be aware that an outcome in the guest is not an absolute guarantee of the same outcome in the original. If an exploit or problem arises from kernel or memory state, it is possible that the hypothetical guest run may miss it and the real host run may fall victim. This seems unlikely to be a common case, given that more problems arise from faulty configurations or buggy executable files than from non-determinism in execution, but it is a possibility and it could potentially be exploited by malware purveyors.

While these initial limitations are serious, there is a mitigating factor. While the theoretical goal is to create an exact duplicate of the entire host, this is a practical impossibility for a variety of reasons. In the first place, the host and guest both make use of the same physical resources as governed by the VMM. This necessarily results in some internal kernel differences. Secondly, the guest can never possibly replicate exactly the host's memory mapping, since it must use less memory and thus fewer mapped addresses than the host. The best case scenario is that the host has memory mapped only into small enough addresses that all will exist on the guest. But, clearly, this is unlikely and certainly it is not a valid assumption. And, finally, the guest will never have exactly the same kernel and memory state since it is not in

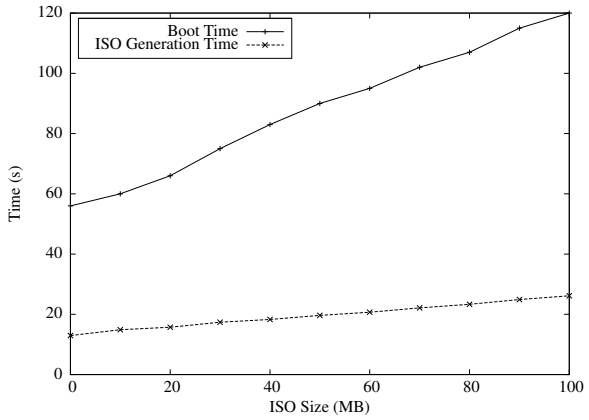


Figure 4: Both ISO creation time and QEMU boot time vary linearly with the size of the host filesystem. Users experience predictable slowdown as the size of their filesystems grow.

the process of replicating itself. In fact, if we allowed exact duplication to occur in the truest sense, we would have an infinite recursion of guests creating their own guests.

So the question of fidelity is not an absolute one of whether or not it exists, but a relative one of what degree we can achieve. Certainly, there is room for improvement here. However, we feel that a duplicate OS with a nearly identical filesystem and the same running user processes is a very reasonable facsimile.

### 4.3 User downtime

At the outset of this effort, we set forth as a goal that the user experience no noticeable downtime. In this regard, we came up well short. The final OSXerox implementation has a downtime that varies with the size of the filesystem and is quite noticeable (see figure 4). There are two main sources of this downtime: the time it takes OSXerox to create the ISO and the time it takes QEMU to boot the guest and launch the host's running processes.

The cause of downtime for ISO creation is reasonably straightforward. In order to create an ISO image containing the host system filesystem, it is necessary to create a copy of all files intended to be placed in the target. Creating file copies, even only in memory, is on the critical path, causing execution times that will be noticeable to the user, particularly as the size of the files to be copied grows.

The other and more significant source of user downtime is the time it takes QEMU to load and boot the guest OS. Upon boot, QEMU emulates the bootloader and unpacks the guest filesystem to be loaded into guest memory. At best, this process can be expected to take as long as it would to boot the original Tiny Core CD – an amount of time already noticeable to the user. In practice, the time for this boot scales linearly with the size of the filesystem being unpacked. And, following the boot, the loading of each running process into memory will also take at least as much time as it did on the host.

After significant development and research, we think that our initial goal (no noticeable downtime for the user) was, perhaps, overly ambitious for such a short time period. When launching a copy of his OS, a user can reasonably be expected to wait the same amount of time it took him to boot and to launch the processes he has running. This would probably, in retrospect, have been a better initial benchmark for downtime.

While this benchmark is the best that can be expected for an ISO based implementation, there is one notable means for improvement on it. A successful implementation of a memory copy scheme could, theoretically, be implemented in the time it takes to launch a user process. This is discussed further in the future work section.

## 5 Future Work

While we feel that the proof-of-concept of a live self-copy is a worthwhile contribution, there are several improvements that we believe could significantly enhance the viability of the OSXerox concept. We think that with more time it is possible to introduce significant improvements in flexibility, usability, performance, extensibility, and device support. The specifics for each are detailed, respectively, in the following subsections.

### 5.1 Flexibility

In its current state, OSXerox has a module that handles collecting user space host processes and describing them asynchronously to the guest. As it exists today, this is accomplished through a series of manipulations of the results from querying the host about running processes. Part of this manipulation involves hard-coded exceptions that should not be run in the guest. For example, XVesa is listed as a user process in the host, but is automatically run by session login in the guest. Duplicating this process in the guest shuts down X, which is an undesirable result.

In subsequent versions of OSXerox, we feel that it would be an improvement to allow process blacklisting via a meta-data file. For any OS, the particular collection of running processes and their invocations is likely to change from time to time. A meta-data approach would allow post-deployment configurations to be adapted rather than requiring new versions.

## 5.2 Usability

As the amount and importance of meta-data grows in our utility, so does the burden on the user to manage it. A worthwhile contribution to the project would include a deploy-time wizard that guided the user through setting up this meta-data. Ideally, it could be re-run at any time as needs and configurations change.

Another helpful usability addition would be automated checks for sufficient memory and memory management in general. In its current incarnation, an attempt to run OSXerox with more than half of available memory in use results in a failure of the VM. It would be preferable for a more polished, deployed version of OSXerox to preempt the user in such a situation.

## 5.3 Performance

Arguably the most significant piece of future work for the project is to implement a memory-copy implementation. As mentioned in the results section, we feel that this could provide a significant boost to minimizing the downtime experienced by the user as well as improving the fidelity of the OS copy. One of the biggest technical challenges that we faced during the course of our work was creating a mechanism that allowed direct, synchronous communication between the host and guest systems but did not involve significant alterations to the source code of QEMU or Tiny Core, thus decreasing the system's broadness of appeal and usefulness.

We feel that the best way to proceed in this area would be to create an ISO image that contains the bare necessities for loading the Tiny Core kernel and a kernel module. This module could open up a channel for TCP/IP communication with the host, block all kernel activity, and receive memory pages. The host would negotiate some rudimentary protocol with the kernel module in the guest and begin pushing over memory pages in stop and copy fashion. Once the communication was complete, the last activity of the host would be to instruct the guest on where to put the program counter to resume guest operation.

Alternatively, there are less ambitious potential sources of improvement. There are optimizations that could be performed both to the ISO generation and to the boot process. A daemon could run on the host that constantly maintains a copy of the host filesystem according to the files specified by the OSXerox meta-data. This would significantly cut down on the amount of time required for ISO generation. Another possible enhancement would be to deploy a core ISO along with OSXerox so that it need not mount the boot disk and copy it locally. And, for speeding up the boot process, a guest could be booted with a user-space module for receiving information from the host and hidden during operational downtime. When the user invokes OSXerox, instead of booting QEMU, it could simply copy the files and process invocations to the guest's waiting receiver process.

In general, the flexibility of our approach allows for a lot of potential performance optimizations. This was one of the main design considerations, given our relatively short development time frame. With the core concept in place and fully functional, performance improvements will follow naturally.

## 5.4 Extensibility

Another important goal of our design is to allow for extensibility. The focus of our work has been entirely on proof-of-concept in the Tiny Core environment using QEMU. However, we feel that the concept is easily ported to other VMMs and OS. Given that ISO is a well accepted standard [14], using a different VMM would be a rather simple matter of installing it on the target and configuring the deployed application to use it. While we have not yet experimented with this, the VMM command line invocation is configurable as meta-data and should be portable to an arbitrary VMM that is capable of booting ISO images and being run from the command line.

Porting the application to a different OS would be somewhat more involved, but we feel that this is also quite feasible. For different distributions of Linux, configurable process filtering and file copying allows a post deployment manipulation of the target files for the guest, including the asynchronous communication for launching processes. For non-Linux operating systems, the extension would be somewhat more involved as the C++ code would need to be ported to and compiled in that environment. Additional constructs would also be necessary, such as configuration of the Windows registry. However, we believe that the general concept is universal enough to be adapted to such OS.

## 5.5 Device Support

As it exists right now, OSXerox works only with a pure-memory OS. However, the addition of another module could extend this to support hard drives. This module would follow the same blueprint as our in-memory filesystem duplication for reading filesystem information, but instead of packing it into an ISO, it would create a disk image for the VMM to use. While this detracts a bit from extensibility given the variation in VMM hard disk representations [5], making use of an OVF appliance would preserve some of the universality. Taking this approach would mandate some additional work in creating and adapting the appliance based on the target VMM.

In addition to hard drive support, the support of duplication for other devices would make OSXerox more robust. However, since VMMs tend to share rather than simulate most other hardware [21], this is not a trivial undertaking. We do not, as of this time, have any specific design suggestions for how best to accomplish such a simulation or how best to negotiate the sharing.

# 6 Related Work

## 6.1 Live Virtual to Virtual Migration

Migrating a VM from one physical host to another has become commonplace, thanks to some important previous work in this regard. One of the earliest such efforts was [4]. While previous works such as Collective [19] and Zap [16] had addressed the problem of VM migration in general, these earlier solutions involved a significant amount of downtime. The general idea was that the fidelity of the copy was paramount and that significant downtime would be acceptable. [4] introduced the notion of iterative pre-copy, which allowed a live migration with unprecedented speed.

Following this work, [3] extended the effort to further reduce downtime of processes and services in the VMs being migrated. It did this by introducing a stochastic model for migrating processes and creating a methodology for scheduling. [9] built on previous efforts by demonstrating that it was possible, in some cases, to improve on migration times by making use of the post copy technique. Previous efforts had consigned themselves to pre-copy and stop-and-copy approaches only.

The effect that this line of research had on our development was effectively to provide a rigorous framework that we could use, but with looser requirements. For migrating live VMs, particularly servers, creating a minimum of downtime is of paramount importance. In our case, minimizing downtime is important, but our only stakeholder is a user and his experience of downtime on a local machine. This gave us more flexibility than our predecessors who had to minimize downtime for connected processes.

## 6.2 OS and Process Migration

In addition to work focused specifically on VMs, we have also drawn inspiration from generalized copy and migration schemes. The aforementioned Zap creates pods, which are process groups with a virtualized view of the system on which they reside. Because of this grouped decoupling from the underlying system, the pods can be migrated with a checkpoint-restart mechanism and not worry about maintaining themselves in two places at once. While the cleanup of residual state is not important to our work, the idea of grouping and migrating processes did have some impact on our thinking about possible approaches.

An interesting work that took place early on in the arena of OS migration is NomadBIOS [8]. NomadBIOS predates the V2V schemes that we discussed and is, itself, a hybrid of OS migration and V2V migration. Based on the L4 micro-kernel architecture, its main feature is the ability to run several instances of Linux on the same physical machine but it also provides the ability to migrate these Linux instances across the network very quickly. While this is interesting in terms of what it implies, its usefulness to us was largely contextual as we were focused on allowing in-place virtualization of commercial OS with no underlying assumptions about kernels or hypervisors.

One of the most important influences for our work was performed by Hanson and Jul [8]. V2V solutions tend to concern themselves with services remaining up for as long as possible while taking for granted the virtualization of the target. Zap [16] and other schemes for migrating resources [17, 23, 11, 6] do not account for the migration of an entire OS. [8] , however, investigates the idea of using the OS as the basic unit of migration and leaving the guest entirely responsible for migrating itself. In doing so, it allows for flexibility in terms of allowing the source system to handle security, copying, and network protocols. While this could represent a drawback in the VM world, it was extremely helpful for our purposes.

If we think of our physical machine as the *source* guest, this addresses exactly what we need to do. It is a scheme completely independent of an underlying hypervisor, and thus could easily be conceptually adapted to an OS running on bare hardware. In their scheme, Hanson et. al. make use of a combination of pre-copy migration and a checkpointing algorithm. In our investigation into memory-copy methodology, we investigated modifying this to make use of stop-and-copy.

## 6.3 Physical to Virtual Conversion

There are various existing Physical to Virtual (P2V) solutions in the commercial sphere. Two solutions we found focus on conversion of an existing physical OS installation to a virtualized copy of the installation; however, they require the system to be shut down [26, 27].

VMWare provides a tool which performs hot cloning, the migration of an OS from a physical machine to a VMM running on a different machine without a system reboot [26]. However, this was of limited use to us since the largest difficulty that we faced was creating and populating the new VM on the source machine itself. This is a proprietary tool so we do not know how exactly the

cloning is performed.

### 6.4 Virtual-Machine Based Rootkit

Another consideration became apparent to us only as we were working on our implementation. In SubVirt [12], the authors explore the creation and installation of a malicious VMM. The idea is that attackers can gain a significant advantage in the security arms race by inserting their code below the level of the OS, rendering traditional malware detection techniques useless. The VMM creates a replica of the user's original machine as a guest in newly inserted hypervisor and it also creates a separate *attack OS* that runs in parallel. This latter OS carries out malicious activities such as hosting phishing servers and interposing on the original OS to evade detection.

The first major influence that this had on our thinking was to introduce us to the concept of Virtual Machine Inspection (VMI), which is an integral part of SubVirt. The second influence was to bring to our attention the ramifications that our work could have in terms of security. While the ability to "go virtual" could be invaluable to users, it could also be devastating in the hands of an attacker. SubVirt requires, on some level, a bit of social engineering to get the user to install the software and reboot the machine so that it can take over. A successful installation of our software would mean that an attacker would simply need to manage to execute our utility in order to virtualize the OS without the user's knowledge.

### 6.5 Memory Migration

Methods for migrating memory are examined in detail in the frame of our work in the Design section. Related work on *Pre-copy* methods [4], *Pure demand-migration* [28], and *pure stop-and-copy* migration [19, 13] influenced our design.

## 7 Conclusion

We have described OSXerox, a tool for creating a virtualized copy of a running operating system and running the copy on the same system. We have shown that the time required to create the copy and boot the system is directly related to the size of the filesystem of the source system. We have also described a design for a live copy of memory pages from the source machine to the guest VM resulting in an exact replica of the source system on itself.

We implemented OSXerox on a Tiny Core Linux system using QEMU as the VMM. Our implementation proves that our design can be used to create a virtualized copy of the running system and run it on the source machine without shutting down. OSXerox is implemented in such a way that it can be configured to operate on other Linux installations, keeping the system flexible and portable. In our evaluation we have shown, that in order to begin working with the virtual copy of their machine, users can be expected to wait for about the same amount of time as it takes the system to boot.

## References


[1] Attacks on virtual machine emulators. Technical report, Symantec, Inc, January 2007.

[2] T. Burger. Development of software for virtual machines. Intel Software Network, October 2008.

[3] F. Checconi, T. Cucinotta, and M. Stein. Real-time issues in live migration of virtual machines. 2009.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[5] Open virtualization format white paper 1.0.0. Technical report, Distributed Management Task Force, Inc., February 2009.

[6] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[7] Analysts examine the growth in server virtualization during gartner symposium/itxpo. Technical report, Gartner, Inc., October 2009.

[8] J. G. Hansen and E. Jul. Self-migration of operating systems. In *EW 11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 23, New York, NY, USA, 2004. ACM.

[9] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 51–60, New York, NY, USA, 2009. ACM.

[10] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *SYSTOR*, page 7, 2009.

[11] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 105–106, New York, NY, USA, 1987. ACM.

[12] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. Subvirt: Implementing malware with virtual machines. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 314–327, Washington, DC, USA, 2006. IEEE Computer Society.

[13] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. *Mobile Computing Systems and Applications, IEEE Workshop on*, 0:40, 2002.

[14] Microsoft. Disc formats. http://msdn.microsoft.com/en-us/library/aa364836(VS.85).aspx.

[15] Mkisofs tool manual. http://linux.die.net/man/8/mkisofs.

[16] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, 2002.

[17] M. L. Powell and B. P. Miller. Process migration in demos/mp. Technical report, Berkeley, CA, USA, 1983.

[18] Qemu. http://qemu.org/.

[19] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, 2002.

[20] Scite project page. http://www.scintilla.org/SciTE.html.

[21] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufman, first edition, 2005.

[22] System restore disk usage. http://support.microsoft.com/kb/300044.

[23] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the v-system. Technical report, Stanford, CA, USA, 1985.

[24] Tiny core linux. http://tinycorelinux.com/.

[25] Vmware vcenter server. http://www.vmware.com/products/vcenter-server/.

[26] Vmware vcenter converter. http://www.vmware.com/products/converter/.

[27] Xen manual p2v process. http://wiki.xensource.com/xenwiki/XenManualPtoVProcess.

[28] E. R. Zayas. Attacking the process migration bottleneck. In *SOSP*, pages 13–24, 1987.