RZ 3778 (# Z1005-003) 05/18/2010
Computer Science 12 pages

Research Report

OS Streaming Deployment

D. Clerc, L. Garcés-Erice, S. Rooney

IBM Research – Zurich
8803 Rüschlikon
Switzerland

Research
Almaden • Austin • Beijing • Delhi • Haifa • T.J. Watson • Tokyo • Zurich

OS Streaming Deployment

David Clerc, Luis Garcés-Erice, Sean Rooney
IBM Research, Zurich Laboratory 8803 Rüschlikon, Switzerland
E-mail: {dcl,lga,sro}@zurich.ibm.com

Abstract

A network deployment of generally available operating systems (OS) can take in the order of tens of minutes. This is prohibitive in an environment in which OSs must be dynamically and frequently provisioned in response to external requests. By exploiting the fact that in general only a small part of an OS image is actually required to be present to perform useful tasks, we demonstrate how an OS can perform work shortly after a deployment has begun. This requires the insertion of a streaming device between the operating system and the disk. We have implemented such a device for Windows* and Linux*. We show that such an *OS streaming deployment* reduces significantly (i.e., to a few seconds) the time between the start of the deployment and the moment at which the OS is available. Furthermore, we demonstrate that the performance overhead of using the OS during streaming is negligible as the penalty introduced by the streaming device is minor and the I/O performance is completely dominated by the multiple caches between the application and the disk.

1 Introduction

Since the standardization of the Pre Boot execution Environment (PXE) [2], it has been possible to deploy operating systems over the network onto x86 architecture machines. Commercial products have built on PXE to load and execute dedicated environments for OS configuration and installation. These environments enable a great deal of flexibility in OS deployment, allowing the automated installation of an OS image over a large number of heterogeneous machines. An alternative approach to OS installation and maintenance of large numbers of clients is the use of a Storage Area Network (SAN). In a SAN, the OS image remains resident on a logical disk on the server, but the disk may be accessed over the network by the client machines, where it appears as a local disk. This method of provisioning machines is termed *OS streaming*, and has typically been used with diskless workstations. The standardization of iSCSI [14] has allowed this to be achieved with commodity servers and clients over any IP network.

SANs have the advantage of fast availability over a traditional network deployment as the OS may boot using the remote disk on the server as soon as the association with the server has been established. The relative performance of using a remote disk over a local one depends on their respective disk bandwidths and the network latency. For a sufficiently fast server disk and network, remote disk accesses may actually be faster than local ones. The main disadvantage of SANs is that the client is always dependent on uninterrupted access to the server. Network connectivity and server availability are required at all times for the correct functioning of the clients. In addition, SANs suffer from a problem of scalability: as the number of clients grows the server infrastructure must be correctly dimensioned to support the increasing load.

We describe a hybrid of these two approaches that we term *OS streaming deployment*. In such a deployment, the sectors of a logical disk in the server are transmitted across the network to the client when they are read for the first time, and then stored locally on the client's disk. Further accesses to these sectors are then obtained from the clients' local disk, removing load from the server. The stored sectors are laid out on the client's disk following the layout of the original image on the server. When all sectors have been copied, the client is no longer dependent on the server: all accesses to the disk are local, and the machine can be rebooted from the contents of its own disk. The end result is identical to a traditional OS deployment, but the OS is available for use during the deployment itself. Our implementation makes use of a streaming block device driver that controls the transmission of sectors from the server and their storage on the local disk. Once the machine has been deployed, all traces of the streaming deployment driver are removed

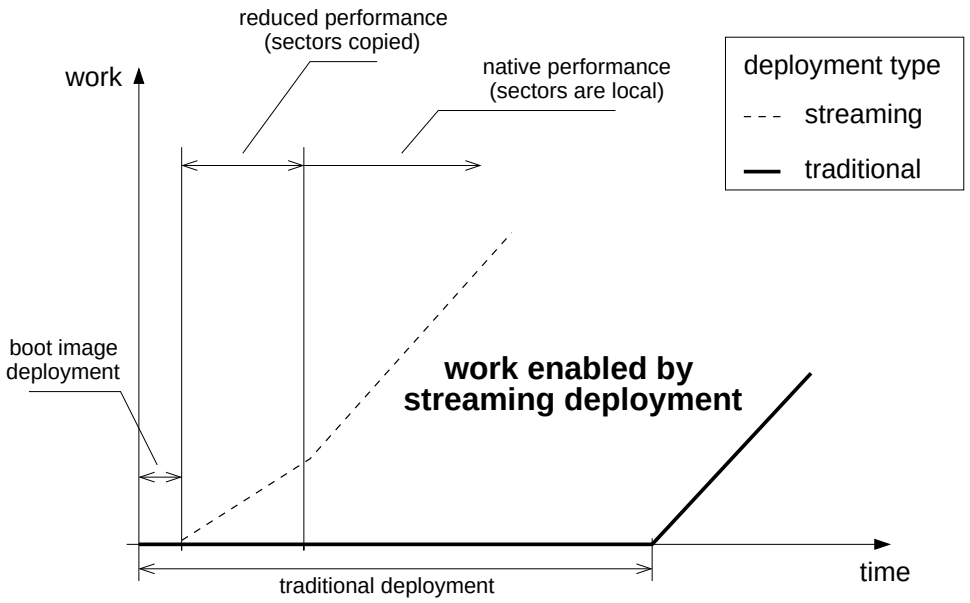


Figure 1: The earlier utilization of resources with streaming deployment results in more work done faster.

and the OS may be rebooted from its own disk containing the original image.

First, in Section 2 we motivate the benefits of streaming deployment by measuring the number of distinct disk sector accesses required to perform various application-level tasks. From this, we can quantify the benefit of OS streaming deployment for a set of representative scenarios. In Section 3, we describe our implementation of an OS streaming deployment system covering both the implementation of the driver for Linux and Windows and the means by which an OS image is prepared for streaming at the server. The driver introduces some additional processing with regard to accessing the disk, in Section 4 we report on the cost of this overhead using both a worst case disk access pattern and industry benchmarks. Finally, Section 5 provides an overview of related work in the field of OS streaming.

2 Motivation

An OS streaming deployment decreases the time during which a machine is unavailable by enabling the execution of the operating system shortly after the deployment has begun. Figure 1 presents graphically the advantage of using a streaming deployment. Fast deployment is most beneficial in an environment in which images are deployed frequently, for example in response to user requests in a cloud computing environment. While many streaming solutions are available for virtual machines, e.g. [16], the novelty of our approach is to enable streaming for physical as well as virtual machines. Thus, we can perform a streaming deployment of the hypervisor itself, provided that the virtualization platform supports the streaming driver (e.g., KVM [8], included in the Linux kernel, or Hyper-V [7] on Microsoft Windows). In production environment often a mixture of physical and virtual machines are used. While virtualization has many advantages, it also has costs in terms of performance and software management. In addition, for commercial Virtual Machine Monitors (VMM) there is also a monetary cost.

We compare an OS streaming deployment with both a conventional networked deployment and with a pure remote boot SAN environment. In the former case the end result is the same, i.e. a fully deployed operating system is running on a physical machine. The latter case is not qualitatively the same as the client can never disconnect from the sever. However, it is interesting to compare with a streaming deployment as it is an alternative method of attaining fast availability.

2.1 Benefit over a Conventional Deployment

The benefit of a streaming deployment over a classical deployment is governed by the fraction of the disk that needs to be accessed before useful work can be performed. Copying less of the disk before being able to execute tasks has two distinct advantages. First, if the operating system is short lived or if its function is limited then it may be the case that much of the disk is never accessed and copying the unaccessed sectors would sim-

ply have been a waste. Second, during a classical deployment the resources are under used as the machine is mainly blocked waiting on I/O and the processor is idle. In a streaming deployment the operating system can be used to fully control and exploit the resources of the machine during the deployment itself.

In order to quantify the benefit of a streaming deployment, we measure the total number of distinct 512-byte sectors that are accessed from the start of the machine boot until the point at which some application-level task is running.

We use an installation of RedHat Enterprise Linux (RHEL) Server 5.5, totaling 4 GB. Among other things this particular OS image features a web server (Apache 2.2.3), a mail server (using Sendmail 8.13.8 for SMTP and Dovecot 1.0.7 for POP), and a database (PostgreSQL 8.1.18). These are examples of applications that are typically provided as services in the cloud. All applications are left with their default settings. We only customize the server installation to ensure the applications above are selected and accessories like games and image manipulation programs are not installed.

Table 1: Disk sectors needed to run certain applications and processes, as a percentage of the original image.

Application	**Sectors used (%)**
Kernel + RAM disk	0.16
Login (Gnome)	7.81
Web server	8.22
Mail server	8.09
Database	8.27

Table 1 shows the percentage of the total sectors from the original image that are required for the different applications to run from the moment the machine boots. Each application is run in an independent boot of the machine, so that one application cannot benefit from sectors accessed by another. In addition, we show the size of the Linux kernel and the initial RAM disk, these together form the minimum image required to boot Linux. The number of blocks accessed until the user is logged into the machine is also given. The login takes place through the Gnome graphical interface as run-level 5 is the default for RHEL 5 Server.

It can be seen from Table 1 that most of the sectors required by many commonly used application are accessed during the boot process (i.e., after the kernel is loaded and until the log in occurs). That is the time where all the OS services are loaded, and the most important system libraries are accessed for the first time. Even when using a GUI, less than 8% of the image is required. Actually running the application adds less than 1% of sectors to the total accessed. Note that the number of the sectors does not include any data, as we consider that the applications are either deployed in a state where one can *start* using them, or they generate data as part of a larger system.

Our experience with Windows 7 yields similar results: from a default installation image of 5 GB, 0.7% is needed for the machine to start booting, and 4% of the sectors are required to log in.

To summarize, for many applications only a small part of an OS image needs to be present to execute. Moreover, a streaming deployment can start doing useful work (booting the machine) after a small fraction (0.16% of the total in the measured RHEL 5 case) has been copied to the client machine. The transfer of this small fraction corresponds to the "boot image deployment" phase in Figure 1. Note that, given that this boot image is the minimum required by the OS to boot, a streaming deployment is arguably the fastest way to boot an initially empty machine on its own OS: we have measured the time required to go from the bare metal to having RHEL 5 boot to be less than 30 seconds (see Section 4.1 for a description of the experimental setup). The absolute lower bound for any *conventional* deployment system is given by the time required to transfer the OS image to the client machine over the network. By way of reference, we measured this absolute lower bound as being around 390 seconds on the same OS and infrastructure (using `dd` over an SSH tunnel, the performance bottleneck being the client's disk). The absolute lower bound does not correspond to actual OS deployment times, as many other functions such as device insertion, configuration, software installation are required to install a generic image on a heterogeneous set of machines. These often necessitate the loading of a reduced image (e.g., Windows Pre-installation Environment) before the target image is installed as the manipulations performed require high-level knowledge of the OS's structure. For example, report [12] benchmarks the installation of Windows Server 2008 on high-end hardware in the absolute best case at more than 28 minutes. This corresponds to our experience with other commercial products: deploying the aforementioned RHEL 5 image on the machines of our experimental setup took roughly half an hour. This deployment is much simpler than the one described in [12], but is also performed on much less powerful hardware. The lower performance of conventional deployments with respect to the lower bound seems to be caused by the transferring of data at the file level, instead of at the disk sector level, and requiring the machine to reboot several times to configure the OS.

2.2 Benefit over a SAN Remote Boot

In a pure SAN environment, all disk accesses are transported over the network. The performance is in large part governed by the file system cache at the client. An application that can frequently be served from the cache will have performance characteristics similar to that of a streaming deployment, whereas applications that often need to read/write from disk will produce a larger load at the server. The *Filebench* benchmark suite [10] allows the workload of a wide class of networked applications to be emulated. We use the Linux port of the 1.4.8 version of the Filebench benchmark to give an indication of the reduction in server load when a local disk is available for storing already accessed sectors in addition to the memory cache. From the set of available workloads we select:

- *varmail*, which is a multithreaded extension of the well known Postmark mail-server benchmark;
- *oltp*, which emulates a database server handling online transaction processing;
- *webserver*; which is inspired by the SPEC benchmark for webservers.

Table 2: Disk accesses over the network in 512-byte sectors

	Total reads		Total writes	
Workload	Average	CoV	Average	CoV
varmail	734	56%	4,740,000	3%
webserver	317,000	48%	425,000	15%
oltp	250,000	5%	1,640,000	5%

We use the default setting in all cases and measure the number of disk sectors that are read or written over the network using iSCSI. The client machine is Redhat Linux 5.4 with a standard ext3 file system.

Table 2 shows the total iSCSI accesses transported to the server after running the test for a duration of 10 minutes and given to the first three significant figures. The results are averaged over ten runs. The caches are emptied at the end of every run of the test. There is a large degree of non-determinism in the tests as witnessed by the Coefficient of Variation (CoV). The total file-set for *varmail* and *webserver* is approximately 16 MB (1000 files with average size of 16 K), which comfortably fits into the 500 MB of RAM available on the test machine. This can be seen in the number of disk reads measured for *varmail*. In the mail server, data is written before it is read and hence is always in cache. The webserver logs information at some frequency, meaning that the large log-file must be read and written across the network. The online transaction processing benchmark performs many small read and writes on very large files that do not fit into RAM. This again is reflected in the read figure. The normalized disk bandwidth required per client at the server can be inferred from the figures in the table. For example, for *varmail* approximately 4 megabytes per second must be written to the server's disk. Assuming a high performance server disk with a bandwidth of 100 MB/s, this would limit the number of clients to fewer than 25. This ignores other factors, such as the network bandwidth, the cost of iSCSI packet processing, the interference between clients etc.

The scalability of the SAN server could be enhanced if the clients were equipped with a local disk such that application data was stored there. To an extent that is what an OS streaming deployment does, except that the existence of the local and remote disk is hidden from the OS by the streaming device and that the contents of the remote disk are replicated on the local disk as and when necessary. The performance of a streaming deployed OS becomes less like that of an OS on a SAN and more like that of a conventionally deployed OS over time.

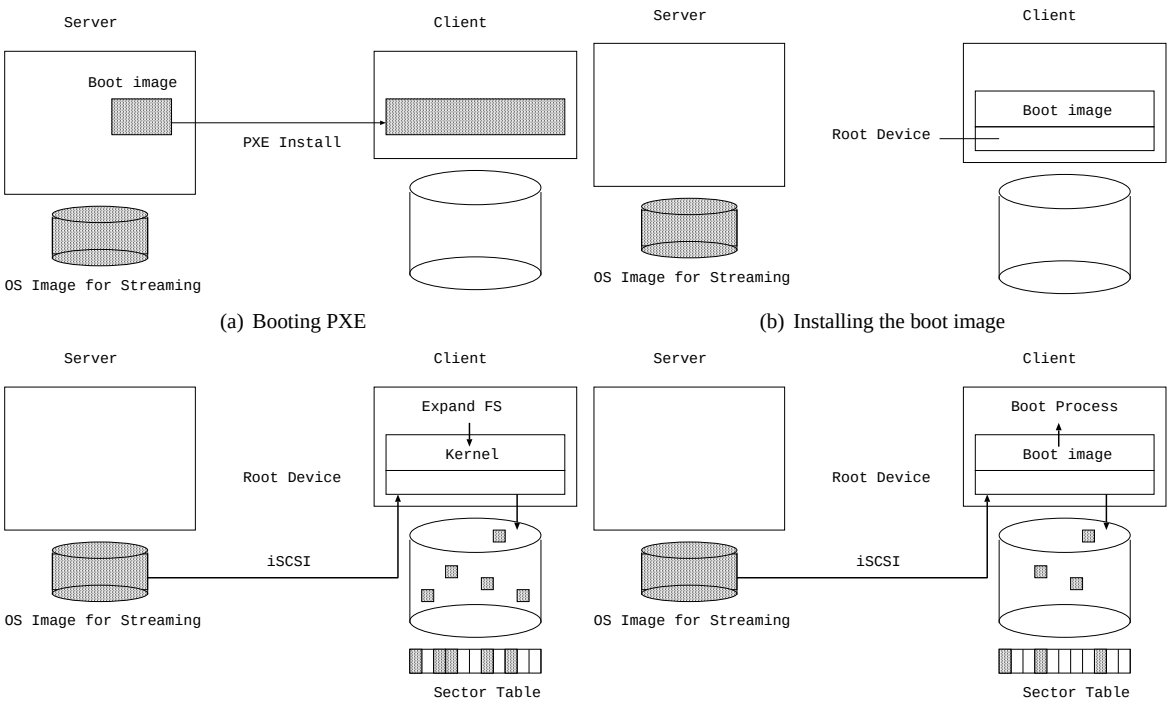
3 Overview

The streaming deployment process is divided into four distinct phases: *PXE boot*, *boot image installation*, *disk initialization*, and *OS boot*. First we detail the activities that take place in each of these phases. Then we describe the way in which an OS image must be prepared prior to streaming. Finally, we outline our implementation of the streaming device. Figure 2 shows the phases of a streaming deployment.

3.1 Streaming Deployment Process

PXE boot See Figure 2(a). In this first phase, the client does a PXE boot, using DHCP to obtains its IP address and the address of a TFTP server. The client downloads a deployment agent from the TFTP server and executes it. The deployment agent prepares the machine during the pre-boot phase, e.g., by writing the iSCSI configuration to the standard BIOS location. The agent is a small (several hundreds of KB) program that runs in the BIOS environment.

Boot image installation See Figure 2(b). The agent can be instructed remotely to download a small boot image corresponding to the OS image to be deployed on the machine. The boot image contains, among other things, the kernel of the OS of choice, device drivers that suit the machine (network card, etc.), an iSCSI initiator, and the streaming device. The agent downloads the boot im-



(a) Booting PXE

(b) Installing the boot image

(c) Initializing the disk

(d) Booting the OS

Figure 2: The distinct phases of a streaming deployment

age (writing it to disk if necessary) and executes it, effectively starting the booting of the OS. Early in the OS boot phase, the iSCSI initiator is started and the streaming device is installed as the OS's root device.

Disk initialization See Figure 2(c). The file system on the root device is then customized so that its size corresponds to that of the local disk. By performing this customization over the device itself, all the file system meta data is read and thus is copied to the local disk. All modern filesystems support resizing (e.g., ext3 on Linux and NTFS on Windows). Other disk preparation operations may be performed in this phase, for example, creating partitions or the installation of a bootloader. A sector table is created on disk that records which sectors are available locally and which still have to be copied. The streaming device only creates this sector table if it does not already exists; otherwise it reads the existing table to determine which sectors had already been copied during a previous session.

OS boot See Figure 2(d). In phase four the OS completes the full boot process. The first read of a sector on the streaming device is carried across the network from the image stored on the server using iSCSI. The sector is then stored locally on the disk at the same position it holds on the remote image, and the sector table is updated. Further accesses to this sector are served locally. At the end of this phase, the OS has booted and all the necessary data are available on the local disk.

After the boot is completed, the operating system is available for use, although only the sectors already accessed are actually on the local disk. Starting applications provokes transfers of additional sectors from the remote image to the local disk. To speed up the completion of the deployment, the system may copy sectors opportunistically when the client and server would otherwise be idle. When all sectors have been copied, the system is in the same state as if it had been conventionally deployed. Note that a full deployment may not ever be needed if all the data necessary for the machine to perform its current tasks are available on the local disk. The deployment could end at that moment, and the machine could reboot autonomously and continue doing the same work without server dependencies. Determining that no single further sector from the server image is ever going to be needed is a difficult task that falls outside the scope of this paper. A possible solution would be to obtain the list of sectors in use by the filesystem (filesystem bitmap), thus determining when all allocated sectors have been transferred to the local disk.

3.2 Boot Image preparation

Generally speaking, an image is the source material used by a deployment engine to create a fully working OS on the target disk. An image contains the bits of data (consisting of an OS, and perhaps applications and/or data) that need to be written on the target disk, but it can also contain metadata needed to create or prepare the target machine. Prior to the streaming deployment, an image must undergo a preparation process. This does not result in the image being modified more than what is required in a conventional deployment (e.g., network configuration). In addition, a small boot image is extracted from the original image to be deployed, following a number of steps that are dependent on the OS.

For Linux, we create a customized initial RAM disk from the initial OS image. This RAM disk includes the necessary drivers for the hardware of the target machine, and the streaming deployment driver and scripts. Together with the unmodified kernel contained in the image, these two elements form the boot image. Whereas within Linux the boot process separates naturally into two phases, i.e., the initial part using the RAM disk and the switch to the final root device, this is not the case for Windows. We define the initial part of the Windows boot process as the period when the network is not available; at the end of this period, the network is available and we can use the streaming device as the root device. We must create the equivalent of the RAM disk manually, by determining which sectors are accessed during the initial part of the boot process and ensuring that these sectors are present on the physical disk before the OS boots.

This is achieved by analyzing the configuration of Windows to determine the list of files that the Windows kernel loader will need to load the kernel and the streaming driver in memory, and mapping this list of files into a list of disk sectors. This operation is performed in a special environment called Windows Preinstallation Environment (PE), with read access to the image to be analyzed. The Windows PE environment is loaded in memory through a PXE boot (ramdisk boot). Windows PE is a minimal installation environment with reduced services that is used to install modern versions of Windows.

The necessary drivers for the client machine can be obtained from an inventory in which the machines characteristics are stored; alternatively, the deployment can take place in the actual physical machine so that the hardware can be detected. Note that in the latter case, although an actual deployment is needed, this is required only once per *type* of machine and that a data-center usually features many units of a few machine models. In the worst case, all drivers in the Linux kernel can be packaged into a single initial RAM disk, ensuring that the machine, if supported, finds everything it requires.

3.3 The Streaming Device

The streaming device is part of the initial boot image and is installed as the root device after network connectivity has been established and before the first access to the local disk. The streaming device intercepts both control and data operations for the local disk. A *sector table* contains the information about which sectors are currently available on the local disk. For read operations, the device determines using the sector table if the required sector is already present on disk. In that case the sector is simply read from the local disk. If the required sector is not present on the local disk, the streaming device requests its transfer across the network. Once the data has arrived the device stores it on the local disk, updates the sector table, and returns the contents of the read sector to the application. All further reads of that sector will take place locally. Write operations are always exclusively applied to the local disk, and never written back to the server. If the written sector has never been accessed before, the sector table is updated to indicate that the sector is now available locally. The sector table is conveniently implemented as a bitmap, such that each bit indicates whether a 512-byte sector is present on the local disk. Hence there is a ratio of approximately 4000:1 between the size of the disk and that of the sector table; so that for for a Terabyte disk, for example, we require 250 Megabytes to hold the sector table. Owing to memory allocation constraints in the kernel, it is not feasible to hold the entire sector table simultaneously in RAM. Instead, we use a paging system such that the sector table is written at the end of the local disk and fixed-size chunks of the sector table are paged in and out as required. The choice of a bitmap for the sector table over more compact representations is driven by the need for paging. The state of a given sector is located at a given location on disk and the *worst-case* cost of reading that state is fixed, i.e., the cost of paging out an in-memory chunk to disk and paging in the chunk containing the required part of the sector table. The amount of memory dedicated to the paging system is dictated by the standard caching tradeoff between size and performance. In Section 4 we perform a worst case analysis and give the performance overhead for actual configurations.

We choose to use the SAN protocol iSCSI [14] for reading sectors of the remote device over the network as there is wide support for this protocol on all targeted platforms. Recently gPXE [5] has added additional protocol support within the pre-execution environment, in particular it is possible for the BIOS to initiate a remote boot using iSCSI. In this way the boot image itself may be deployed using iSCSI and the streaming device could use the remote disk without having to include an initiator in the boot image. The transfer of state, e.g., the IP address

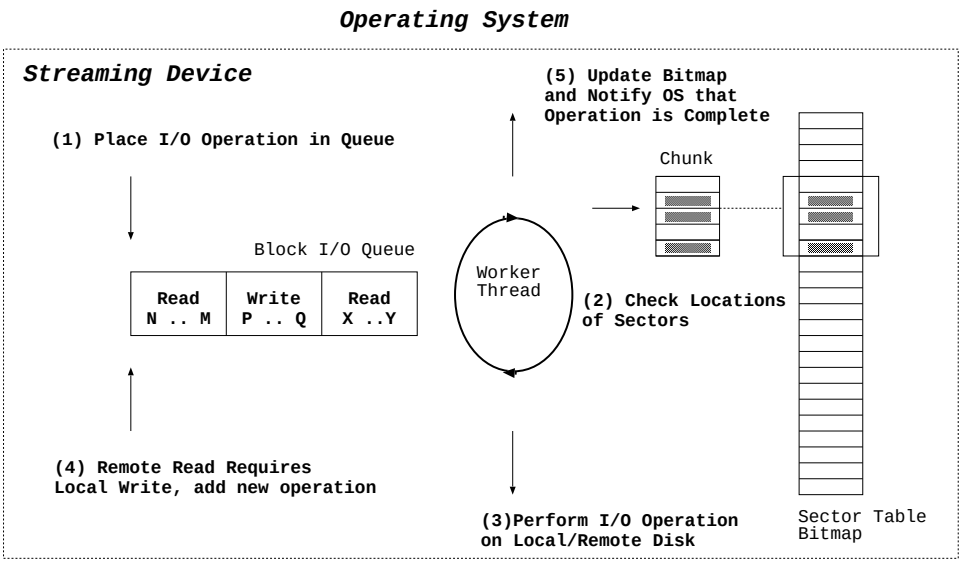


Figure 3: The streaming device driver

of the iSCSI server, between the pre-boot and boot phase is enabled by the standardization of a table containing this state, the iSCSI Boot Firmware Table (iBFT, as defined in [3]), which may be written during the pre-boot phase for the OS to read.

In the system described the streaming device copies sectors from a single server. Unlike in a conventional SAN it never writes to the logical unit at the server, meaning that with suitable extensions, the same disk can be shared among many clients. Moreover, the client may get different parts of a disk from different servers, if the servers all have identical copies. This requires that auxiliary iSCSI targets are identified using some additional management process in addition to that identified in the iBFT. Moreover, clients that implement an iSCSI target can serve disk sectors to other clients. Interestingly, the distribution of an OS image can then be treated in a similar way to that of a large file in a BitTorrent like file sharing system. These ideas are further developed in paper [13].

We have implemented such a device for the Linux 2.6 kernel and Windows 7. Although the frameworks for Linux and Windows are rather different, there are similarities in the constraints imposed by the kernel environment, leading to a common programming model. In both OSs, the device driver works as a router for I/O operations between the iSCSI disk, the local disk and the filesystem driver above. I/O operations received on the streaming device are submitted to the appropriate disk for processing and a callback is executed upon completion. These callbacks are required to execute without blocking in both Windows and Linux, meaning that they are extremely limited in the computation they perform: they cannot perform other I/O operations, allocate memory, or even read from the sector table as this may provoke paging. In consequence, we have an event-driven model in which callbacks generate events that get placed in queues and a dedicated worker thread executing in process context reads these queues and performs the required computation.

Figure 3 summarizes the activities of the device when the OS requests that a sector I/O operation be performed on the disk.

The Linux driver makes use of the device-mapper framework that has been part of the kernel since version 2.6. The device-mapper allows operations on one block device to be mapped to another. Our *dm-stream* loadable module implements the device-mapper interface. This device takes as parameters the names of the local disk device and the iSCSI device, and uses the device-mapper framework to issue operations on them. The boot script inside the RAM disk uses the device-mapper to make *dm-stream* visible as a mapped device; this mapped device is then set as the root device.

The Windows driver is implemented as an *Upper Class Filter Driver* for the disk class within the Windows Driver Framework model. The Windows model allows the filter driver to discover the iSCSI and local device through the Plug and Play (PnP) facility.

4 Performance

The use of the streaming device driver implies some overhead when compared with using the local disk directly because a sector must be read across the network and written to disk the first time it is accessed. Moreover, it is necessary to check the local availability of a sector on every read access, which in turn may involve paging out a chunk of the bitmap to disk and paging in another. We follow the guidelines given in [15] and quantify the actual cost of using the device driver in two ways.

First, we construct a test which gives the worst-case overhead for the driver when compared with native access, i.e. when it is necessary to page in and out chunks of the sector table on every disk operation. Note that this is the *worst case* in a relative sense and not the worst case in absolute throughput: we measure the largest *penalty* incurred. In fact, as we shall see, the read throughput for the worst-case test pattern may, under certain circumstances, be greater than reading through the file system.

In addition, we report the performance difference between running an industry file system benchmark over the native disk and over the streaming device driver.

4.1 Test Setup

All experiments are performed in the following infrastructure: We set up an iSCSI target on a blade server with 2 Intel* Xeon* processors at 3.20 GHz with Hyper-Threading, 1024K L2 cache and 2 GB of RAM running RHEL5. The iSCSI target is IET 1.4.18. The target exports an OS image (i.e., a file) on a SCSI disk (model Seagate ST973401LC) as an iSCSI disk. The client blade is similar to the first, but features 2 Xeon processors at 3 GHz with Hyper-Threading, 512K L2 cache and an IDE disk (model Fujitsu MHT2040AS). We believe that this setup reflects the most typical scenario of an image being exported by a server having more powerful storage than the client. The two blades are connected through a Gigabit Ethernet switch.

4.2 Worst-Case Streaming Driver Overhead

The least favorable sector access pattern for our mechanism is the one that triggers the need for a new chunk of the sector table bitmap at every access. To generate this pattern, we access one sector at intervals whose length is equal to the number of sectors contained in a fixed-size bitmap chunk. When the sector is accessed for the first time, it must be copied from the remote disk and the in-memory bitmap chunk is updated. When the next sector in the series is accessed, this triggers the writing to disk of the previous chunk (as it has been modified) and the retrieval of the next. Consequently, during the first pass, on every disk access, we must read and write an entire chunk. On successive passes over the same sectors we do not need to update the bitmap so no writing of the chunk is required, and no network access need be performed.

To perform the experiment we use an iSCSI initiator on the client blade running a Fedora 11 (kernel 2.6.29) with our streaming driver. For simplicity of analysis, we configure the device to allow only a single chunk in memory. Allowing more simultaneous in-memory chunks would reduce the I/O overhead, but increase the kernel memory usage of the device.

The resulting measurements are shown in Table 3. Column *First access* shows the I/O throughput measured when accessing the sectors for the first time. This shows that the performance for both the local disk and the streaming device is low, e.g., less than 5 MB/s for all chunk sizes. There is no noticeable difference between the performance of the streaming device and the local disk, because in this configuration the disk cache plays no role, and performance is driven entirely by seek times; disk seek times are two orders of magnitude slower than the network latency.

In Column *Steady state* shows the I/O throughput measured when sectors are accessed repeatedly. In this case, the local disk provides good performance, e.g., more than 100 MB/s. This is because of the on-device disk cache (8 MB according to the disk specification). For the streaming device, the performance penalty in can be as high as 56% (32 KB chunk). In this case, most of the work being done is retrieving and writing the sector table chunks from the disk (each 4 KB of read data requires 8 times as much meta-data to be retrieved). The performance penalty is reduced to $\sim$11% when the chunk size is 4 KB.

For a specific disk access pattern we could configure the chunk size and number of chunks such that the overhead of paging is negligible. For example, when there is high locality in disk accesses, large chunk sizes will be beneficial. If there is a small number of distinct locations on disk that are often accessed, then choosing a number of chunks equal to that number will give good performance. The figures reported show that even when the paging configuration is entirely inappropriate for the access pattern, the penalty is no more than a factor difference over native access. These measurements do not allow us to infer anything about the I/O throughput that an application will observe. In the next section, we examine the performance of the driver in more realistic scenarios by using an industry benchmark.

Table 3: Performance of worst-case reads over the device with normal access (MB/s).

	Local disk		Streaming device	
Chunk size (KB)	First access	Steady state	First access	Steady state
32	4.58	112.42	4.20	49.28
16	3.41	129.65	4.63	70.64
8	3.13	138.26	4.72	95.62
4	3.40	113.72	4.05	100.69

Table 4: Operations tested with iozone

Operation	**Test description**
Write	Write a file sequentially
Read	Read an already created file sequentially
Rewrite	Write an already created file sequentially
Reread	Read an already read file sequentially
Randwrite	Write to a file at random locations
Randread	Read from a file at random locations

4.3 File System Benchmark

We use the *iozone* micro benchmark [11] to compare the performance of a file system mounted on the driver against mounting it directly on the disk. We choose *iozone* because it is simple and available for both Windows and Linux. *iozone* is designed to allow the detection of performance bottlenecks, it measures a range of file operations performed on files of varying sizes and using records of varying length. It also allows the tester to control the options used when performing the operation, e.g., synchronous against asynchronous writes. In all cases we use the default options for *iozone* and select some subset of file operations; Table 4 summarizes these operations. Note that as a file is always written before it is read, all sectors of the file are written locally and then read locally, i.e. the remote device is never accessed and what is tested is the overhead of having to go through the device before accessing the local disk. We only consider files which are too large to fit into the L2 data cache of the processor (512 KB).

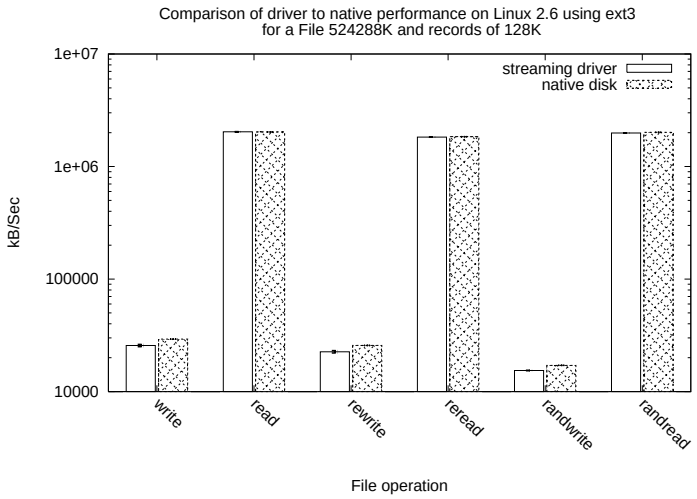
Figure 4 shows the cost of performing various file operations for Windows and Linux on a file of size 524 KB that fits comfortably into the 2 GB of memory available, but which is large enough to require multiple levels of block indirection within the inode. The micro-benchmarks are highly deterministic, and the standard deviation is less than 1% of the average in all cases recorded.

The performance of the different types of *read* are governed by the file system cache and are identically for both the native and streaming device. For *write* operations, the native disk allows faster throughput than the driver although this is never more than 10%. We only show the results for a fixed record length of 128 KB because while additional caching effects are observable for different record sizes, e.g. file system read ahead, both the native and the stream device benefit from them equally.

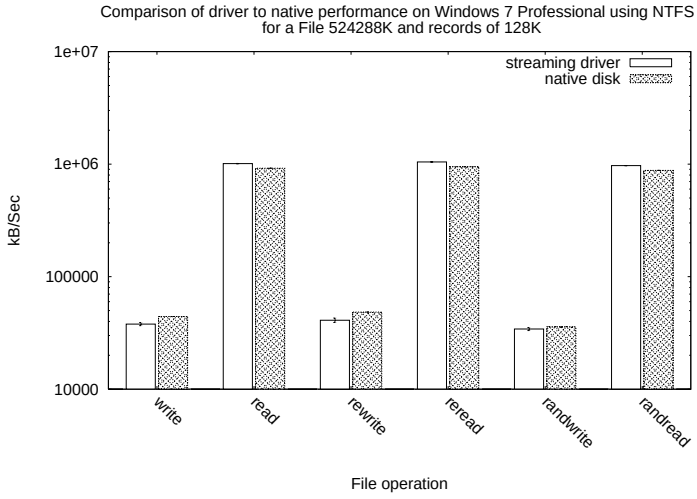
Interestingly, in the Windows case, the writing performance seems to be better than when using Linux, whereas the reading performance is better for Linux. These different behaviors are governed by different caching policies and filesystem design decisions in the operating systems. Otherwise, the results for both Windows 7 and the Fedora 11 show the same picture with respect to the overhead introduced by the driver.

Figure 5(a) and 5(b) show the cost of performing various file operations on a file of size 3 GB that cannot fit into memory. We give a detailed view showing how the performance changes across a range of record sizes. We omit the results for Windows because, as Figure 4 showed, it behaves very similarly. The performance of the *write* operations is similar to the previous case, with the native disk doing slightly better than the driver. The *read* performance is much less than that in the previous case because the benefits of the cache disappear and reading and writing are broadly similar. The *random-write* is worse than sequential *write* for small records sizes, presumably because the disk head must move much more. As the record size increases, the overhead per byte of moving the disk head reduces and *random-write* then approaches the performance of *write*. The *random-read* behaves similarly, but with the difference that for a *random-read* there is some probability that the record sought is in the cache, whereas for a sequential *read* the probability is zero, as the file is bigger than the cache and when repeatedly read sequentially the cache always misses. Consequently, as the overhead of disk head movement reduces with increasing record size, the benefit of caching starts to manifest itself and the *random-read* actually out performs the sequential *read*. This effect is observed for both the driver and the native case.

Also note that the observed read throughput for both the native disk and device driver is significantly less than that reported in Section 4.2 when the disk was being accessed directly. This shows the overhead of accessing



(a) Linux 2.6 results



(b) Windows 7 results

Figure 4: Performance of iozone micro-benchmarks on a file that fits into the system cache

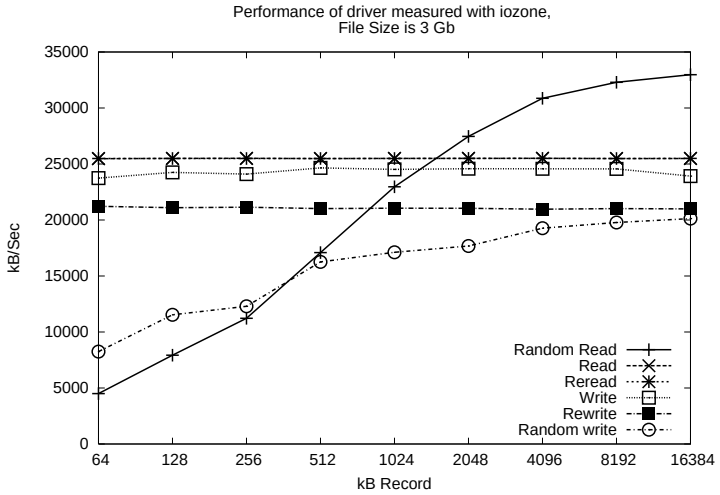
the sectors of a large file via the file system when many meta-data blocks must be read to identify the data blocks.

In summary, the observed performance of file I/O is governed mainly by the caches at the file-system level (and above). The overhead of using the driver can be observed for file write operations on large files, but the effect is less than 10% on all observed cases. We emphasize that the penalty of using the driver is only experienced during the OS deployment, subsequent to this all disk accesses will be native.

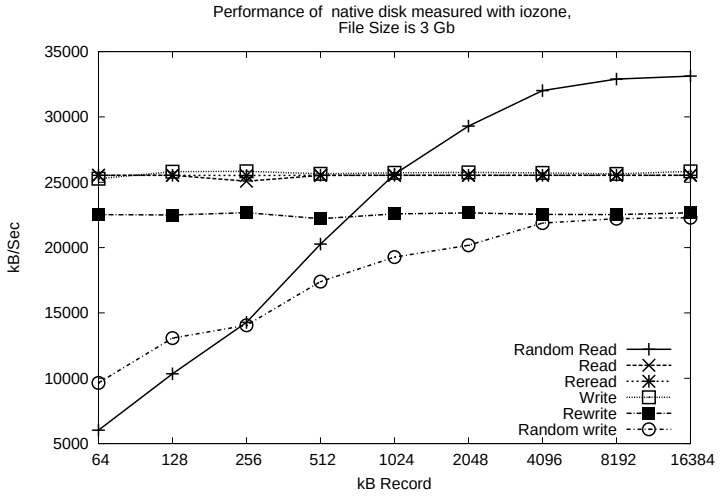
5 Related Work

The *dm-cache* system described in [6] allows sectors that are accessed from a storage area network to be locally cached on disk so that further accesses to those sectors are performed efficiently. It is a Linux-specific solution and make uses of the device-mapper to create a driver. dm-cache uses its own caching format to lay sectors on disk, meaning that it is never possible to disconnect from the server. The totality of the cache meta-data is retained in memory, i.e. there is no equivalent of our sector table paging system. For dm-cache, this is a not a concern because the cache is small relative to the size of the disk and the cache contents can be lost without affecting the functioning of the system.

The Citrix provisioning server [9] allows multiple clients to boot off a virtual disk (vDisk) held on a server. The client may be running on either virtual or physical machines. The clients contain a special device that allows transparent access to the remote virtual disk.



(a) Driver



(b) Native Disk

Figure 5: Performance of iozone micro-benchmarks on a file that does not fit in the system cache (Fedora 11 Linux 2.6 results)

Clients have only read access to the vDisk, allowing multiple clients to share the same image. Write access to the vDisk is achieved by storing written blocks separately; this cache of written blocks may be held at either the server or the client itself. The objective of the Citrix provisioning server is not to physically deploy OS images but to simplify their management; consequently the server can never be removed.

iBoot [1] enables a machine to boot from a remote disk. This approach allows centralized management of the disks. iBoot is thus marketed as a disk-less workstation solution. iBoot consists of a ROM image, which contains iSCSI client code, a TCP/IP stack, and BIOS interrupt code. Upon boot, the BIOS disk I/O interrupt goes through the iBoot code to communicate directly with the remote iSCSI target, providing seamless access to the SCSI logical units. iBoot is now part of the firmware of IBM's PowerPC* based blade servers.

CCBoot [18] is a means of creating a disk-less version of Windows based on a combination of iSCSI and an extended form of PXE — gPXE [5]. gPXE extends the standard configuration of the BIOS to support iSCSI as well as other protocols. This means that Windows can be configured to boot *transparently* from a remote disk. gPXE can either be flashed in the ROM of the native BIOS or installed on an externally pluggable device. Alternatively, PXE can be used to install gPXE, which then can modify the PXE environment. The latter is the mode that CCBoot uses.

VMWare has a number of technologies using stream-

ing methods. *ThinApp* [17] delivers application data to a computer as the application is being executed. The interest of ThinApp is being able to run applications without having to install them or running a virtual machine. The applications may require changes in the registry, environment variables, etc, these are performed in a sand-box in which the application runs rather than in the VM itself. Paper [4] propose something similar for physical machines enabled by redirecting Windows' system calls. After the application has completed no trace is left behind. ThinApp uses streaming to load the applications from a shared repository. *VM Streaming* is a feature introduced in VMWare Workstation 6.5 [16] that enables booting a VM from a virtual disk by providing its URL; the contents will be downloaded as needed by the execution. As in essence the virtual disk is a file, VM Streaming copies the file across the network, such that it is ultimately available on the machine on which the VM is executed. Our work also allows the streaming of a virtual machine, but unlike VM Streaming it is independent of the Virtual Machine Monitor used. Moreover, when kernel based hypervisors such as KVM [8] or Hyper-V [7]. are used our system enables the streaming deployment of the hypervisor itself onto the bare metal.

6 Conclusion

We have shown how operating systems can be made available for use during their deployment. This is of increasing importance in environments in which many different types of OS are deployed in response to user-driven demand. Our solution is to perform a streaming deployment in which sectors are transferred on their first request. Compared with a conventional network deployment, this decreases the time during which the machine is unavailable by an order of magnitude. We have reported on our implementation of an OS Streaming deployment system, describing the means by which it is implemented on two widely available operating systems, namely Windows 7 and Linux 2.6, and provided relevant performance results.

References

[1] iBoot: Remote boot over iSCSI. `http://www.haifa.ibm.com/projects/storage/iboot/index.html`.

[2] *Pre Boot Execution Environment (PXE) Specification, v2.1*, September 1999. `http://www.intel.com/design/archives/wfm/downloads/pxespec.htm`.

[3] *Advanced Configuration and Power Interface (ACPI) Specification, Revision 3.0b*, October 2006. `http://www.acpi.info/spec30b.htm`.

[4] ALPERN, B., AUERBACH, J., BALA, V., FRAUENHOFER, T., MUMMERT, T., AND PIGOTT, M. PDS: a virtual execution environment for software deployment. In *In Proceedings of the 1st International Conference on Virtual Execution Environments* (Chicago, IL, USA, June 2005), ACM Press, pp. 175–185.

[5] ANVIN, P., AND CONNOR, M. x86 Network Booting: Integrating gPXE and PXELINUX. In *Proceedings of the Linux Symposium* (Ontario, Canada, July 2008).

[6] HENSBERGEN, E. V., AND ZHAO, M. Dynamic policy disk caching for storage neteworks. *IBM Research Report RC24123* (November 2006).

[7] KAPPEL, J. A., VELTE, A., AND VELTE, T. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., New York, NY, USA, 2010.

[8] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the Linux virtual machine monitor. In *Proceedings of the 2007 Linux Symposium* (Ottawa, ON, Canada, June 2007), vol. 1, pp. 225–230.

[9] MADDEN, B. A better way to manage Citrix servers: centralized block-level disk image streaming. *Citrix White paper* (March 2006).

[10] MCDOUGAL, R. Filebench: a prototype model based workload for file systems, work in progress. *Sun Microsystems*. `http://www.solarisinternals.com/si/tools/filebench/filebench_nasconf.pdf`.

[11] NORCOTT, W., AND CAPPS, D. *Iozone Filesystem Benchmark*, March 2010. `http://www.iozone.org`.

[12] PRINCIPLED TECHNOLOGIES, TEST REPORT COMMISSIONED BY DELL. *Time Comparison for OS deployment, Dell United Server Configuator version 1.1 vs HP SmartStart version 8.25 x64*, August 2009.

[13] ROONEY, S., AND GARCÉS-ERICE, L. Parallelizing OS deployment through streaming. *Submitted as a Brief Announcement to PODC 2010* (2010).

[14] SATRAN, J., METH, K., SAPUNTZAKIS, C., CHADALAPAKA, M., AND ZEIDNER, E. *Internet Small Computer Systems Interface*. Network Working Group RFC, April 2004.

[15] TRAEGER, A., ZADOK, E., JOUKOV, N., AND WRIGHT, C. P. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage 4*, 2 (2008), 1–56.

[16] VMWARE, INC. *Virtual Machine Streaming*, August 2008. `http://www.vmware.com/products/beta/ws/releasewwws_ws65_beta.html`.

[17] VMWARE, INC. *Streaming Execution Mode – Application Streaming with VMware ThinApp*, June 2009. `http://www.vmware.com/files/pdf/VMware_ThinApp_Streaming_Execution_Mode_Information_Guide.pdf`.

[18] YOUNGSSOFT. CCBoot diskless boot WinXP/Win2003/Vista/Win2008. *Youngssoft User Manual* (October 2009).