Abstract

OS Streaming is a common data center technique for deploying an OS image quickly onto a physical or virtual machine in which the machine requests the individual blocks of the image from a server as it needs them. When streaming images the server's OS level block cache brings very little in terms of performance as the collection of images is usually too large to fit in memory. We investigate how to improve the scalability of streaming servers by ensuring that blocks shared among multiple streamed images are preferentially retained in a deduplicated cache. We outline the nature of our deduplicating block cache, describing how cacheable blocks are identified during an offline deduplication process and how an extended form of the Least Recently Used (LRU) block replacement algorithm can be used within the server cache.

1 Introduction

A computer uses a collection of software to perform tasks. This collection is usually formed by an operating system (OS), which provides the interface to the underlying hardware, and applications, which use the OS to run on a given machine. Such a software collection can be stored in the form of an operating system image, or simply image, which is a representation of a disk with the OS and applications already installed.

Images are usually large, each of them containing Gigabytes of data. They are stored in some repository at a central location, whose simplest form consists in having a file per image on a storage device such as a disk. For a machine to make use of one of these images the image contents must be copied over the network to the disk of the machine. Given the large size of the images, this is a costly operation that takes a long time even on powerful hardware. The preferred method for transferring the image contents is on-demand, using a streaming approach. As a result, portions of the image that are not needed for a particular scenario are not transferred to the disk of the client machine. Mechanisms to stream virtual machines are presented in [12] and to physical machines in [5].

While memory I/O and network I/O are measured in Gigabytes and Gigabits per second respectively, disk I/O is measured in tens or hundreds of Megabytes per second. Moreover, the announced disk I/O figures are typically for the absolute best case, i.e. reading-in-place, with minimal disk rotation and seeks. If filesystem blocks are read randomly from disk, then a reduction of one or two orders of magnitude in throughput is to be expected. It is clear that in a streaming system, it is of critical importance to reduce disk I/O to a minimum.

Observing the patterns of I/O at streaming servers reveals some distinct features. First, there are typically no writes at the server as the writes are performed locally on the client, for example using a copy-on-write overlay disk and permanently stored (if necessary) using ulterior means. Second, ideally the data that was read by a given client will be stored by the client in its block cache or the local disk and will rarely be read from the server a second time. Third, certain patterns on the disk will be predictable, for example booting a given OS up to a login prompt is deterministic on the operating systems we have traced. Fourth, there is much redundancy between images as they share many blocks in common.

Work in the literature, e.g. [10, 1, 7] has proposed exploiting the commonality between images to reduce the total amount of storage required. This process is called deduplication. While deduplicating on disk reduces storage, it can significantly reduce the effective disk I/O. We propose a system that uses similar techniques to those described in the literature but with a different objective: to increase the scalability of the streaming server by improving its block cache. The number of images in use in a data center explodes as users produce more and more images by customizing a collection of initial images. By exploiting deduplication we ensure that identical blocks of different images are only read from disk and stored in the cache once, and are then usable by multiple different clients streaming multiple differ-
2 Background

Deduplication is the means by which data items that are present in a data set are stored only once. Duplicate data items may be recognized by creating a digest from the content of the data item, such that the digest with high probability uniquely identifies the data content. When a data item is deduplicated, the digest is calculated and it is determined if the same digest has already been seen. If so the item is a duplicate and the data content need not be stored a second time, if not the digest is stored in an index. Digests are conveniently calculated using hash-functions, e.g. SHA-256. Such content addressable storage is used in commercial deduplicating system, e.g. EMC Corporation’s Centra appliance.

2.1 Efficiently Storing Many OS Images

Two approaches exist for reducing the storage allocated to OS images. The first is to keep a small number of reference images and reconstruct the required larger number of images from separately held deltas [2]. Similarity is identified by making sketches of a file, which capture some feature of it and then assuming files with similar features have similar content. This is most appropriate when only a small number of root images are supported within the environment and that the delta themselves are never large, e.g. there is only a small amount of configuration applied to the root images.

The second approach, deduplication, is better when there is no obvious relation between the images stored. When deduplicating an OS image, it can be presented as simply a single opaque file and deduplicated using block-level deduplication or an attempt can be made to mount the filesystem in the image and deduplicate the files held within it.

The latter case is more complicated as it requires the system performing the deduplication to posses the same filesystem drivers as the operating system in the image it is deduplicating, however it potentially allows a more compact deduplication as large files can be deduplicated as a single entity. The block-level deduplication of operating system images is better when recovering the original image quickly is important as the storable and bootable forms are similar. For example a virtual disk image can be stored as a file on a deduplicating filesystem e.g. [11] and booted directly from that filesystem. From the hypervisors view point it is a normal file containing an image, while the filesystem ensures that duplicate blocks are only stored once.

Within a deduplicating filesystem larger block sizes can be more efficiently read from disk, but they reduce the potential for deduplication. We identify the maximum block size that deduplication allows through the following experiment. From a single Linux* image we create new ones by creating empty images of the same size, formatting them and mounting them, so that the directory structure of the original can be copied into the new images. Files are copied in the same order from the original to each new image. As a result, all images contain exactly the same data (a default installation of Fedora 11 with an ext3 filesystem, the image being roughly 6 GByte in size with 5.2 GByte of data). Then we proceed to deduplicate the images and measure the storage required to store them. We compute the storage required by counting the number of different blocks in all images. A block is different from another if it produces a different output of the SHA256 function. We exclude unused block space from our analysis.

Fig. 1 shows the storage actually used when this is done for 6 images. Ideal deduplication is achieved when an arbitrary number of images with identical content, no matter how it is arranged internally, take the space of one single image. The results show that 4 KByte blocks allow ideal deduplication. For blocks larger than 4 KByte little deduplication is achieved. Recall that these images contain exactly the same data. The explanation for this is that 4 KByte is the filesystem block (fs-block) size on the images in the expen-
ment. The ext3 filesystem does not lay out the data on disk independently of the process performing the copying\(^1\), i.e., the images are not exact copies at the block level although they contain exactly the same data. No matter how ext3 arranges these 4 KByte fs-blocks on disk they can always be deduplicated when the deduplication mechanism also divides the image into 4 KByte blocks. As we increase the deduplication block size beyond 4 KByte there is a higher probability that the fixed-size chunks in which the images are divided are not aligned with the groups of 4 KByte fs-blocks in the filesystem, and so it becomes increasingly difficult to find blocks with the same content to deduplicate. For example, if the filesystem has a block size of 4 KByte, while the deduplicating block size is 16 KByte, no file under the size of 16 KByte will ever be deduplicated except under the unlikely circumstance that the following 12 KByte on the image are identical in two or more images. Even files of size 16 KByte or greater will not be deduplicated if they are not aligned at the same 16 KByte boundaries on all images. Given that modern filesystems do not store the same data deterministically on disk, we must align the deduplication strategy with the fs-block size to achieve significant deduplication. Consequently, when deduplicating operating system images a pragmatic choice of deduplicating block size is 4 KB as this is a lower bound for most filesystem block sizes on mid- and large storage devices.

Also note that Fig. 1 shows that little deduplication is achieved within a single image no matter which block size is used. A single operating system image only contains a small number (less than 10%) of blocks that occur twice or more. We have found this is also true for other operating systems such as FreeBSD and Windows 7.

2.2 Efficiently Running Many OS Images

On a given hypervisor multiple virtual machines may be running. Each of them is using some subset of the pageable memory available. The pageable memory is potentially the most important resource on such a system, as frequent paging in and out of disk will have severe consequences for performance. In a way directly analogous to storage deduplication, benefits can be expected if identical pages of distinct virtual machines can be held only once in the page memory of the hypervisor. Commercial industrial hypervisors such as VMWare ESX server perform the dynamic deduplication of virtual machine pages. Gupta et al [6] report improvements over the performance in ESX in regard to memory usage by the use of additional techniques. In particular, by keeping deltas between similar pages, patching them as required and compressing pages that are infrequently used. They claim that these extremely fine grained optimizations are justified as memory will increasingly be the most precious resource on a hypervisor. Consequently, it is worth trading additional processing cycles against more compact memory usage. Deduplication is performed constantly by the hypervisor in phases that are measured in tens of seconds.

3 Efficiently Streaming Many OS Images

We recall that we wish to stream many images with a significant degree of similarity from a given server infrastructure to many clients in a data center. The clients may be either physical or virtual machines. From the point of view of the server the nature of the clients is unimportant, it simply interacts with them across a SAN protocol, for example iSCSI. The operating system images are stored in a format that the iSCSI server can stream, typically as one or more files stored on the server’s disk.

The objective is to optimize the usage of the cache to minimize the disk I/O. In particular, we never load the same block content into the cache twice. It would be possible to perform this deduplication inline, for example in a way similar to [6] such that after the block is read we determine if it is already in the cache and if so do not store it a second time. However, this would serve no purpose as we would pay the cost for disk I/O the first time any block on the disk image is read, which will only be read again if it is flushed from the client’s cache, and by then it may not be available from the server’s cache. Rather, we need to know which blocks are identical before we read them from disk. Consequently, the deduplication of the images needs to be performed offline.

It is natural to consider using a block-based deduplicating repository such that the images are deduplicated on disk offline and the repository ensures that the same block is only present once in its dedicated cache, i.e. we extend the storage deduplication mechanism to optimize the cache. This could be implemented either as a dedicated repository, using a user-level deduplicating filesystem [1] or a kernel-level deduplicating filesystem e.g. [11]. However, as we have seen, deduplication of operating system images requires a fine block granularity, typically 4 KByte. The result is that a given image is broken into many small pieces. Since many of these pieces will have already occurred in other images and will already be stored on disk, the storable representation of the image will be scattered across the disk. The consequence is that as an increasing number of similar images are deduplicated against one and another that the ac-

\(^1\)Ext\(^*\) filesystems “color” the location on disk where new blocks are allocated with the ID of the process requesting the blocks. Thus, the same installation of the same OS on different machines yields images with different block layouts. Other block allocation strategies in these or other filesystems may introduce more non-determinism on where blocks are finally stored. But even for a perfectly deterministic filesystem, consider that storing one extra file when an image is created may shift the blocks of all the remaining data, breaking all existing alignment above the filesystem block boundaries.
cesses on disk become more and more random. Random accesses on 4 KByte blocks incurs an enormous performance penalty.

We demonstrate the effect on performance of deduplicating in a filesystem using the state of the art: ZFS* developed by Sun*, the Solaris* ZFS filesystem provides many advanced features including deduplication. We can not perform any measurements on Solaris, as this requires explicit permission from Oracle. Thus we use NexentaCore Platform v3.0.1 from September 2010, which is based on OpenSolaris, the open-source version of Solaris. We install the OS on a blade with 2 Intel® Xeon processors at 3.20 GHz with Hyper-threading, 4 GByte of RAM and a 73 GB 10K RPM SCSI disk. ZFS uses a block size of 128 KByte by default, but as we have seen in Section 2.1 we require deduplication to operate on 4 KByte blocks, so we set the block size accordingly when creating the deduplicating filesystem.

We copy the images described in Section 2.1 into the filesystem, picking each time one at random until the filesystem is full. Although adding additional images does not increase the amount of data in the filesystem it does increases the amount of meta-data. For our experiment the filesystem was almost completely full after 69 images had been added. ZFS reports a degree of deduplication close to 60x, meaning that on average each block in the filesystem is referenced 60 times. Given that there are 11 GByte of data outside the deduplicated filesystem, we can say that our 69 images are perfectly deduplicated.

We measure the sustained throughput of the underlying disk to be 58.1 MByte/s (within spec). We then read images in 4 KByte chunks from both the normal filesystem and the deduplicating filesystem and measure the achieved throughput to identify the performance penalty of filesystem deduplication flushing the block cache between runs.

When reading sequentially from the normal filesystem we achieve 43.5 MByte/s, while with ZFS we obtain 22.9 MByte/s, i.e. there is almost a 50% performance penalty for deduplication. This penalty is similar for all the 69 images.

In the experiment described above all images share the same blocks, so all the data is located contiguously on a reduced portion of the disk, even if the meta-data is not. In a more realistic scenario, for a moderately large collection of images, an extra image added to the repository would find most of its constituent blocks scattered all over the disk, contained in the previously stored images. We attempt to replicate a similar situation with the following experiment: given an image, we store all the odd (first, third, etc.) 4 KByte blocks in a file in the empty deduplicating filesystem described before. Then we copy 8 other images containing different OS (various versions of Windows XP, Vista, 7, FreeBSD and Suse Linux). Finally, we copy all the even blocks (second, fourth, etc.) of the original image to another file, and then the image itself. We expect the blocks of the image to be divided between the first file and the second file. And indeed, the performance obtained when sequentially accessing the blocks falls to 13 MByte/s, i.e. 30% of the performance of the unduplicated filesystem. As the blocks get distributed among multiple images and not just two large files, we would expect performance to degrade further.

In summary, we wish to deduplicate offline, but only to optimize the block cache. We do not need to deduplicate the storable form of the image, which can remain as a single large file, which the underlying filesystem will try to ensure is held as contiguously as space allows. Given the abundance and price of storage, the benefit of saved disk space brought by a deduplicating file-system is outweighed by the performance penalty imposed on the server’s disk I/O. In effect, what we require is a deduplicating block cache.

4 Implementing a Deduplicating Block Cache

Koller et al [8] propose a deduplicating block cache to reduce disk I/O at servers. They outline how such a system could be implemented as a kernel level block device and argue through evaluating traces of production systems that it would bring significant performance enhancement. The motivation for our work is similar, but we exploit the fact that the streaming system is not distributing the content of arbitrary files, but that of streamable operating system images.

The deduplicating block cache is dedicated to image streaming and is, in the normal mode of operation, distinct from that of the OS block cache. For convenience it is implemented in user space, using either a user-level filesystem, a user level disk device e.g. SCST or extensions to an iSCSI server. An important advantageous side-effect of implementing the block cache in user space is that the cache is contained on ordinary user process pages. On a demand-page systems, such as Linux, these pages are the last pages to be scavenged when the system is running low on memory. Consequently, the OS block cache or other buffers allocated by the OS will not interfere with the deduplicating block cache. We assume that streaming is the only significant activity going on the server, and allocate most of the machine’s memory to the deduplicating cache.

The number of times that a block appears in a set of images can easily be determined by deduplicating all the images and storing a counter for all the resulting blocks. However, the frequency counter itself would not fit into RAM resulting in an extremely slow process that needs to be repeated on the addition of every new image. Work in storage deduplication has already considered the problem of indexes that do not fit into RAM, for example [3] proposes
using a hybrid solution in which a solid state disk is used to store the image index allowing a much better random disk I/O throughput. Others have proposed using simpler and more compact indexes, such as bloom filters [13] which quickly allow the determination of whether a block has already been seen or a sparse index [9] in which the number of deduplicated blocks is traded against a smaller index.

Having identified the degree of deduplicating of the block in the set of images under consideration we exclude all those blocks with a number of duplicates inferior to a certain number. Blocks which only appear infrequently in the set of images will not bring much benefit by retaining them in cache. By excluding them we can keep the cache more compact. The exact threshold values is determined by the amount of cache memory available.

For each image we keep an image index that maps multiples of a 4 KByte offset to its digest if the corresponding block is cacheable (we assume unallocated disk space is never requested, as it would be trivially cacheable and consequently it would inflate the image index unnecessarily). When a request for a given offset on a given disk image is received, the block cache examines this image index to determine if all or parts of the requested sectors are ones that are cacheable. If not then the corresponding sectors are read from disk as normal. Otherwise, we use the resulting digest to examine the content index to determine if the blocks are already in the block cache, if so they are returned immediately from there. If they are not, they are read from the disk and stored in the block cache (following a block replacement algorithm described in the next section) and the content index is updated.

The memory allocated to the block cache is used to hold both the meta-data and the data blocks themselves. For every unique 4 KByte block held in cache we need to hold the digest at least once, assuming this is N bytes, then the lower bound for the amount of memory devoted to meta data is N/4096, e.g. for SHA-256 it is less than 1%. The actual ratio of meta-data to data is determined by the degree of deduplication and the compactness of the meta-data data structures. Allowing for 5% overhead for the total metadata, we can retain approximately $2.5 \times 10^5$ blocks in cache per GByte of available pageable memory.

### 4.1 Block Cache Replacement Algorithm

A replacement algorithm determines which block should be removed from a cache when the cache is at or near capacity. The reference algorithm in OS Block caches is Least Recently Used (LRU), in which the block that has been accessed the least recently is chosen as the candidate to be removed. LRU is an optimal algorithm if the probability of a block being accessed in the future is similar to that of its past history. For reasons of performance, the algorithms used in modern OSs are only approximations of LRU, for example the WS Clock algorithm [4].

The simple LRU algorithm is not appropriate within the streaming cache for two reasons: First, the server cache is shared between multiple clients meaning that the locality that LRU requires is diminished; Second, not all block accesses are observed at the server. For example, if a given block is frequently accessed at the client it will be retained in the client’s cache and will not be accessed at the server.

Instead, we extend LRU within the deduplicating cache such that instead of removing the least frequently used block, we remove the least frequently used block content. We demonstrate the efficiency of this approach through the following experiment.

We generate synthetic images such that for an image containing N blocks and for $0 \leq x \leq 1$, the number of blocks that occur $2^p$ times is given by $f(p) = N \cdot x(1 - \sum_{i=0}^{p-1} f(i)), p \geq 0$. We let $x=0.9$ meaning that 90% of blocks occur once, 9% twice, 0.9% four times, etc. This approximates fairly closely to the duplication distribution we observe in actual operating system images. Having generated such an image, we model image customization by randomly choosing a fixed percentage of the image and ensure that they are unique within the repository of images. For the experiments we set this image customization at 10%.

We model the image access patterns by having the process randomly access a block which is only a short distance from the one it last accessed, we call this the local mode. Occasionally the reading process will jump to another random location within the image. Two parameters define this behavior: the maximum distance from the current block that the process will move to when in local mode, we define this as 1% of the size of the image; the number of blocks accessed before a jump is performed, we define this as 1% of the size of the image.

We model the cache behavior at the client using a conventional LRU algorithm on a per-client cache whose size is variable, and we report the figures for client caches that are 10% and 1% of the size of the image. The total number of accesses across the entire set of images is the product of the number of images and the number of blocks they contain.

The images are read using two different server cache replacement algorithms. The first is the conventional LRU (Normal) algorithm and the other is a deduplicated form of LRU (Dedup). The reported figures are for server caches that as large as a single image. The figures given show the ratio of cache hits to misses at the server. Figure 2(a) and Figure 2(b) shows the measured performance for the difference server cache algorithms with increasing number of images. For a single image the two algorithms perform almost identically, while with increasing number of clients the normal cache performance degrades while that of the deduplication cache performance improves. The normal
LRU algorithm depends on the locality of access in order to have good performance, i.e. the next block accessed will be close to the last one, as the number of images increases this becomes less and less likely as the server cache is shared amongst all clients. The deduplication LRU algorithm depends on similarity of content to have good performance. With increasingly numbers of clients with similar operating system images there is a increasing chance that another client has recently read the same content from the server in the recent past. Although the amount of unique content increases with number of images (as 10% of each image is unique within the repository), the unique content is rarely accessed and hence is quickly expelled from the deduplicated LRU cache. What is retained is the content that is most frequently accessed, i.e. those blocks which are highly duplicated within the repository.

In the experiment the client cache is always smaller than the server’s cache; so for example for a single client the server acts as an auxiliary cache to that of the client, retaining a longer history of accesses. This benefits the client when not all the block accessed by the client in local mode can be stored in the client cache. So for example, for a single client when the client cache is 1% of the image, 40% of the server accesses can be served from the server’s cache. As the client’s cache increases in size this benefit is reduced, so for example when the client cache is 10% of that of the server, only 25% of the accesses to the server can be served from the cache. If the server and client cache were the same size then the server cache is effectively redundant for the normal cache algorithm. This is exactly what is observed in the experiment.

Figure 2(a) and Figure 2(b) together show that the larger the client cache the worse the normal cache behaves, but the deduplication cache is unaffected.

5 Conclusions

Streaming is the fastest and most flexible way of assigning OS images to machines in a data center, independently of the technology used to allocate resources (virtual or physical). As more and more users run customized versions of these images on shared hardware, an scalable mechanism for streaming operating systems becomes a central piece of the data center infrastructure. We have demonstrated that an OS streaming server can benefit from a deduplicating block cache. We have investigated the constraints in such a system explaining why deduplicating streamable operating systems is distinct from other deduplicating systems. In particular, we have shown how the size of the deduplicated block size is mandated by the operating system filesystem and how storage deduplication has a significant impact on performance. We have outlined how a user-level deduplicated block cache can be implemented and have described how the LRU block replacement algorithm can be adapted for its usage.

* Oracle, Sun, Solaris and ZFS are registered trademarks of Oracle and/or its affiliates. Linux is a registered trademark of Linus Torvalds. Windows, Windows XP, Windows Vista and Windows 7 are trademarks of Microsoft Corporation. Intel and Intel Xeon are trademarks of Intel Corporation. IBM and PowerPC are registered trademarks of International Business Machines Corporation. Other product and service names might be trademarks of other companies.
References