TotalCOW: Unleash the Power of Copy-On-Write for Thin-provisioned Containers

Xingbo Wu
Wayne State University
wuxb@wayne.edu

Wenguang Wang
VMware, Inc.
wenguangw@vmware.com

Song Jiang
Wayne State University
sjiang@wayne.edu

Abstract

Modern file systems leverage the Copy-on-Write (COW) technique to efficiently create snapshots. COW can significantly reduce demand on disk space and I/O bandwidth by not duplicating entire files at the time of making the snapshots. However, memory space and I/O requests demanded by applications cannot benefit from this technique. In existing systems, a disk block shared by multiple files due to COW would be read from the disk multiple times. Each block in the reads is treated as an independent one in different files and is cached as a sperate block in memory. This issue is due to the fact that current file access and caching are based on logic file addresses. It poses a significant challenge on the emerging light-weight container virtualization techniques, such as Linux Container and Docker, which rely on COW to quickly spawn a large number of thin-provisioned container instances. We propose a lightweight approach to address this issue by leveraging knowledge about files produced by COW. Experimental results show that a prototyped system using the approach, named TotalCOW, can significantly remove redundant disk reads and caching without compromising efficiency of accessing COW files.

1. Introduction

Copy-on-Write (COW) is a widely adopted technique for minimizing cost of creating replicas in the memory and on the storage devices, such as hard disks. In modern file systems, such as ZFS [11] and Btrfs [20], COW enables an efficient creation and management of snapshots. Taking Btrfs as an example. When creating a snapshot of a set of files, only their key metadata need to be duplicated. The snapshot is created as a new directory[1]. At this moment, there is only one physical copy of the set of files on the disk. In the meantime, there are two sets of logically independent files presented to users. When the files are modified, only the blocks containing the modifications are re-written at a newly allocated space on the disk, and the other blocks are still shared. In this way, the cost of file duplication is mostly proportional only to the actual number of modified blocks, rather than total size of the files. As a result, creating backups of a large file system can be done in a timely manner with negligible impact on the running applications.

Snapshots have been widely used in the deployment of today's virtualized systems and their efficiency is of great importance. The emerging container-based virtualization platforms, such as Linux Container [5], Docker [2], and Hyper-V Container [9], leverage snapshots to spawn a (large) number of container instances derived from a common container template. In this case, multiple snapshots will be created from the template, which is often a directory containing a root file system. Each container will be booted from its own snapshot and read shared content in the template until it is modified. In this way many commonly used files in the template will likely be shared by a large number of virtual machines. For example, it is common to see a Hadoop cluster consisting of hundreds or even thousands of nodes [7] is deployed from a template containing Hadoop's program package of over 300 MB. Together with other necessary software, the template can exceed 1 GB. By using the container-based technique leveraging COW snapshots to share a few copies, one can save hundreds of Gigabytes of disk space in the system installation.

1.1 The Issue

A file system with the COW capability on the storage device has the advantage of reducing disk space consumption and I/O operations for file duplication. However, when shared blocks in snapshots are read into the memory, this advantages is not retained. Unix operating systems use a common abstraction layer–Virtual File System (VFS)–on the top of

[1] To create a snapshot, the source directory needs to be a 'subvolume' in Btrfs.

APSys '15, July 27–28, 2015, Tokyo, Japan.

http://dx.doi.org/10.1145/2797022.2797024

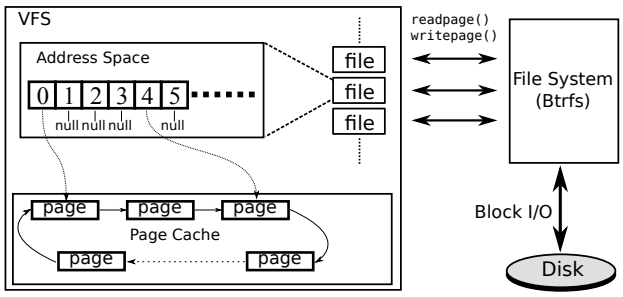


Figure 1: VFS Organization.

various device-specific file systems. Figure 1 illustrates relationship between VFS and its underlying file systems. When an application requests a block of data from a file managed by an on-disk file system, the data block will be read from the disk into the page cache in memory, assuming that they are not in the cache yet, before it is delivered to the application. Interestingly, when a data block has only one physical instance on the disk but belongs to multiple (logical) files due to use of COW, without any knowledge on the sharing VFS cannot use the data block in the memory cached for a different file. Instead, it has to issue a new read request for the block again and allocates a new space in the memory to cache a duplicate copy.

The root cause of the issue is that VFS manages the cache on the abstraction of logical files, rather than on physical disk space [17]. While this is a sound design choice (as some file data may not have been mapped to the disk space yet), it can lead to significant wastage of memory space and I/O bandwidth in the reading of different COW'ed files. In other words, the benefit of a COW file system exists only with the on-disk data, and not with in-memory cached data. This issue can be serious for several reasons. First, compared to the disk the memory is of very limited capacity and is much more expensive. Wastage of memory space would reduce memory available for running applications and lead to more block/page replacements and corresponding I/O operations. Second, repeatedly reading data that have been in the memory from the disk imposes unnecessary workload on the disk. Furthermore, the additional I/O time is often on the critical path of applications' execution, compromising their performance.

1.2 Inadequacy of the UnionFS/OverlayFS Approach

OverlayFS [6] and UnionFS [19] are popular alternatives to the COW file systems for creating COW containers from templates. Arguably they can partially address the aforementioned issue. Unlike a COW file system that copies data and manages copied data at the block granularity, they enable COW at the file granularity. That is, if any part of a file is modified, the entire file is physically copied to produce a new file. For a shared file in the template that is not yet modified and onto which multiple logical files in different virtual machines (or containers) are mapped, OverlayFS and UnionFS use a unification layer to directly map the logical files onto the file in the template. In this case, the pathnames of the logical files are actually symbolic links pointing to the real file in the template, and the unification layer translates a symbolic pathname to the real file's pathname. As VFS can see the real file pathname, it is aware that multiple logical files with different symbolic pathnames but sharing one common physical file are actually a single file. Accordingly, reading the data in the file via different symbolic links would not incur multiple I/O operations and keep redundant data in the page cache. As OverlayFS and UnionFS apply COW at the file granularity instead of at the block granularity, they work well as long as most modified files are small, or not significantly larger than a disk block (4KB). However, it would take a long time to service a small write on a large file, as an entire large file has to be duplicated before the write request is completed. This time is on the critical path of the request service, causing unexpected performance degradation of I/O service. To make the matter even worse, after new files are physically produced possibly due to small writes, all the benefits of COW, including on-disk and in-memory space saving and reduced I/O operations, are lost. With these limitations, the solution provided by OverlayFS and UnionFSare not adequate and cannot be considered as a general-purpose one.

1.3 Our Contributions

To address the issue, we have three objectives: (1) minimizing disk space provisioning using Copy-on-Write; (2) efficient cache sharing to minimize memory footprint, and (3) effective caching to avoid unnecessary I/O operations. We propose a non-destructive approach that requires easy instrumentations of operating system and file system to achieve the objectives. A prototype system is implemented in the Linux kernel with the Btrfs file system. Collectively the design and the implementation are named **TotalCOW** as they retain the benefit of the on-disk COW technique and enable in-memory COW to greatly improve the memory and I/O efficiency of accessing snapshots in a COW file system.

Our contributions in this paper include:

- We identify and analyze the issue on memory and I/O efficiency with the use of COW file systems for container-based virtualization systems.
- We propose a two-level approach to be applied in operating systems to effectively address the issue.
- We develop a prototype system (TotalCOW) and evaluate its effectiveness in term of memory usage and I/O throughput.

The rest of the paper is structured as follows. Section 2 describes the design of TotalCOW, which is evaluated in Section 3 with a comparison with Btrfs and OverlayFS.

We discuss the effectiveness of TotalCOW in Section 3.3. Section 4 describes related works, and Section 5 concludes.

2. The Design of TotalCOW

Efficient copy-on-write of on-disk data has been well supported by some file systems, such as ZFS and Btrfs, as one of their major features. However, VFS uses a common interface, or a set of functions (e.g., `readpage()` and `writepage()`), to interact with different file systems. In VFS, file blocks are cached in the page cache based on their logical addresses in their corresponding files. Once a block is read from the disk into the cache managed by VFS via the interface, all the information specific to a file system, such as the data's disk location, is lost. VFS' interface leaves little room for passing information about COW'ed blocks from file systems to the kernel for VFS to avoid potential redundant reads and caching.

One might intend to revise the interface to allow file systems to report their data on-disk addresses to VFS, which would then organize its page cache according to the data's disk addresses. In addition to requiring significant changes to the kernel, which by itself is not a desirable choice, this approach has several other limitations. First, the mapping from a file page in VFS to its disk location can be changed by the file system at any time (such as for disk defragmentation), and it is a challenge for VFS to keep the information always up to date. If VFS had to consult the file system on every read/write operation on the mapping's validness, file access would become complicated and inefficient. Second, the approach cannot handle the case where file blocks have not yet been mapped to disk addresses, including newly created files in VFS that have not been committed to the file system, and file systems that do not need backing physical block devices, such as NFS and tmpfs.

Without disruptive changes to the existing organization of kernel page cache and the interface between VFS and file systems, we propose a lightweight approach to effectively address the issue of performance loss due to redundant disk reads and the issue of cache space loss due to redundant block caching. For the former issue, we build a new layer of cache that is close to the disk, in which blocks are indexed by disk addresses. For the latter issue, we develop an efficient method to quickly identify blocks shared by multiple files so as to avoid caching redundant copies. In the below, we will describe the two components of TotalCOW.

2.1 Caching with Disk Addresses

As indexing existing page cache with disk addresses is not an viable option, we add a new layer of cache between the block device and the file system. The cache acts as a wrapper of the block device, and any I/O requests issued to the device will be first caught by the cache, whose data are indexed with disk addresses. Because the purpose of the cache is only to remove redundant disk reads that occur on template files

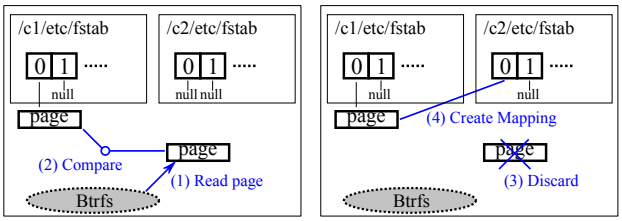


Figure 2: Hint-assisted identification and removal of redundant page in the page cache.

and cannot be detected by the page cache, it only caches the disk blocks belonging to the template files. To this end, the cache maintains a list of disk block address ranges where these files are stored. While only blocks in the ranges can be cached, the list can be considered as a whitelist. We provide a kernel module to allow users to register and deregister template files through `sysfs` [18] into the whitelist. Most file systems, including Btrfs, provide functions to reveal file-to-blocks mapping information, such as using `ioctl()` with flag `FIEMAP`. We use this API to obtain disk block addresses of a file and add corresponding address ranges into the whitelist.

Note that any block accesses to disk addresses not covered by the whitelist would bypass the cache. The cache is a read-only cache—if a block in the cache is written, it is removed from the cache. The size of the cache can be configured and we use the LRU replacement algorithm. While TotalCOW uses additional memory for having the cache, it reduces memory consumption from $O(n)$ to $O(1)$ in terms of the number of common-source snapshots (n is the number of the container instances).

2.2 Hint-assisted Avoidance of Duplication in the Page Cache

To keep the way in which VFS interacts with its underlying file system unchanged, TotalCOW does not attempt to determine whether two blocks in different files are mapped to the same disk block by leveraging metadata that are only available in the file system, such as Btrfs. Instead, upon a read request it allows VFS to request data blocks from the file system, if needed, as usual. Because of TotalCOW's new cache layer, the request can be efficiently serviced. When VFS receives the data blocks, for each block it needs to determine whether a block of the exactly same content has existed in the page cache by comparing its data with data of (some) blocks in the cache. If yes, the newly retrieved block is discarded and the block's logical file address is mapped to the block currently in the page cache. Figure 2 illustrates the steps. A block is added into the page cache only when there is not such a match in the comparison.

A concern in the design is the cost of comparison, which may involve a very large number of blocks for each newly loaded block. To minimize the cost, we limit the comparison within a small number of candidate blocks. When a

number of snapshot files are created from one template file, we call the group of snapshot files/directories as common-source files/directories, and blocks at the same offset of the files as common-source blocks. Accordingly, a block is compared only to its peer common-source blocks. To enable this technique, TotoalCOW must be aware of which files are common-source ones and use it as a hint for determining comparison candidates. TotalCOW provides an interface for users to specify from which template file (directory) a new snapshot is created. It treats all snapshots derived from the template as a group.

As an example, two snapshot directories (`/c1` and `/c2`) are created from one common template to support two containers (`c1` and `c2`). Accordingly, two files in the snapshots, `/c1/etc/fstab` and `/c2/etc/fstab`, are common-source files as they share a common suffix in their namepaths (`/etc/fstab`). Suppose the `fstab` file in Snapshot `/c1` has been read and fully cached in the page cache. When VFS tries to read a page in the `fstab` file in Snapshot `/c2`, it sends the request to the underlying file system to retrieve the block as usual. After VFS receives the block, TotalCOW compares it with the cached common-source block belonging to `/c1`, which is identified with the hint. In this way, the overhead for identifying and removing redundant blocks in the page cache is small.

By removing redundant blocks, multiple (logical) blocks can be mapped to the same block in the page cache. When one of the blocks is modified, TotalCOW creates a private block in the cache for the block to receive the modification. At this time the file system may not be aware of this change until the dirty block is written back to the disk. This design preserves the functionality of the page cache as a write buffer. In this way, TotalCOW provides an in-memory COW in addition to the on-disk COW provided by file systems.

The number of candidate blocks in a common-source block group is equal to the number of snapshots created from a template. If the number is large and many blocks have been modified, the comparison cost can be still substantial. To this end, we attach a one-byte signature to each block, in the page cache, belonging to some common-source snapshots. In the signature, the first bit indicates if the corresponding block is a private one, which has been COW'ed in the memory by TotalCOW. The remaining seven bits are hash value of the block content. For a full-block data comparison, the candidate block must not be a private one, and the 7-bit hash value of the new block and that of the candidate block must match. By conducting the screening, TotalCOW minimizes the chance of unnecessary comparisons. Figure 3 shows an example use of the signatures. Note that a candidate block that is private on the file system may not be considered as private in the memory by TotalCOW because it is re-loaded after it becomes private in the file system.

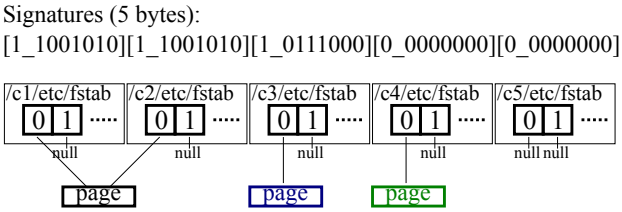


Figure 3: An example use of signatures. There are five snapshots in a common-source group. For a block in Snapshot `/c1`, only corresponding block in Snapshot `/c2` needs to be compared. Snapshot `/c3` has a mismatched hash value, Blocks of Snapshots `/c4` and `/c5` have been modified.

3. Performance Evaluation

In this section we evaluate efficacy of TotalCOW in terms of its performance improvement and memory efficiency. A kernel module is implemented to provide an interface to userspace for configuring snapshot groups. The generic file read/write routines (mainly in *mm/filemap.c*) of the Linux kernel are also modified to hook the core functions of TotalCOW into the kernel. We use the stock Btrfs as the COW file system in the evaluation of TotalCOW.

We compare TotalCOW to Btrfs on the stock Linux kernel for managing the snapshots, and to OverlayFS that creates on-the-fly directories for containers. OverlayFS also uses Btrfs as its underlying file system.

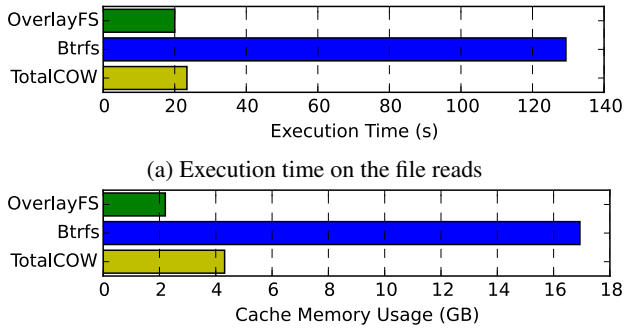
The test machine has two Xeon L5410 CPUs, 64GB DDR2 ECC memory, and two 1.5 TB `WD15EARX` hard drives. We use Linux 3.18.6 for implementation and testing, and LXC 1.1.1 for managing containers. In the configuration of TotalCOW, we do not limit the size of its disk-address-mapped cache to reveal its full potential on removing unnecessary reads on snapshots.

In the below we will first use a read-only workload to evaluate the systems' memory and I/O efficiency, and then use a write-only workload to evaluate the impact of different COW techniques on the write performance. Both workloads are generated using Linux command `dd` and temporary in-memory file(s). We use `dd` to copy on-disk file(s) to the temporary file(s) for generating read-only workload, and use `dd` to copy the temporary file(s) to on-disk file(s) for generating write-only workload.

3.1 Read-only Workload

We evaluate memory efficiency of TotalCOW in a simplified scenario where all containers read their respective snapshot files mapped to a common template file on the disk. The size of each file is 2 GB. The machine's 64 GB memory is sufficient to accommodate 8 duplicates (16 GB) of the file in the page cache. We measure the elapsed time spent on read as well as the final cache usage after the reading is done.

The experiment results are shown in Figure 4. Btrfs has the worst execution time and the largest cache footprint as



(a) Execution time on the file reads

(b) Cache space consumed by the read data

Figure 4: Reading common-source files in eight containers.

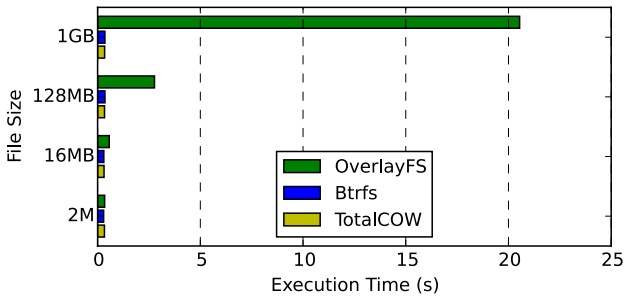


Figure 5: Overwriting 2 MB data in files of different sizes

each container independently reads a copy from the disk and keeps a private copy in system's page cache. OverlayFS produces the best efficiency because the eight containers actually access the same file in VFS. TotalCOW's cache footprint is doubled compared to OverlayFS due to its use of a new cache layer. A page needs to be cached both in the page cache and in the new cache. However, its execution time is only a little bit longer than that of OverlayFS because copying pages within memory is much faster than that from the hard disk. The disk space consumption is minimized in all three systems as all use Btrfs as the file system and no new blocks are allocated with read requests.

3.2 Write-only Workload

In this experiment we use a write-only workload to evaluate efficiency of copy-on-write operations. As discussed in Section 1, a small write in OverlayFS may lead to copying of an entire file, which can cause long stall times for large files. In contrast, Btrfs natively supports COW on the disk at the block granularity, making the I/O cost proportional to the actual amount of written data. TotalCOW relies on Btrfs to handle COW, so it does not incur any additional I/O cost for servicing writes. The only overhead could be that for evicting dirty blocks in the new cache layer, which requires only in-memory operations.

We first set up an experiment to evaluate the performance of making small modifications in a file. Note that the original file is in a snapshot and none of the modifications would be applied directly on the original file. Four files of different sizes {2 MB, 16 MB, 128 MB, 1 GB} are created as the template file in each setting. The execution time is collected by overwriting the first 2 MB data in each file. A `fdatasync()` is called after each test to flush the dirty data to the file system. Specifically, for a 2 MB file the entire file is overwritten, while only 1/512 of the 1 GB file is overwritten. Figure 5 shows the execution times with different file sizes. Both TotalCOW and Btrfs use less than 0.2 seconds to complete the write, regardless of the file size. As we expect, the result shows that a COW file system can efficiently handle write on shared files. In contrast, OverlayFS takes more than 20 seconds to write 2 MB data into the 1GB file. This is equivalent to merely 200 KB/s throughput, far lower than the raw disk bandwidth. Note that this does not mean OverlayFS consistently has such low performance. Once a file has been duplicated, further writes will be as efficient as other systems. In addition to the aforementioned performance issue, the replicated file consumes additional disk space and common-source files can no longer share its blocks in the page cache.

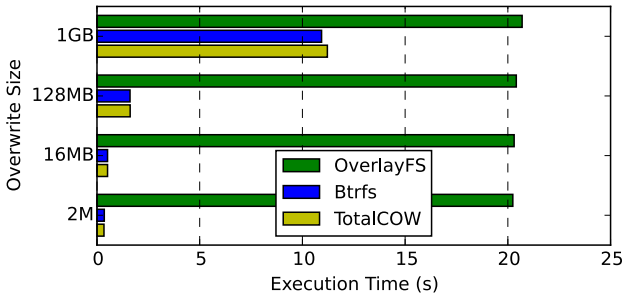


Figure 6: Overwrites with various data sizes on a 1 GB file

We conduct another experiment to investigate the performance impact of writes of different data sizes. In this experiment we fix the template file's size to 1 GB. Data of various sizes (2 MB, 16 MB, 128 MB, 1 GB) are used to overwrite a part of the file. Figure 6 shows the execution times with different overwrite sizes. For TotalCOW and Btrfs, the execution time increases proportionally with the size of write. OverlayFS takes at least 20 seconds to complete a write. Its execution time is almost unchanged with the increase of the write size. An entire file is read from the snapshot file into the memory once the file is opened for write. It is then written to a newly created file no matter how small the fraction of the file for overwriting is. Even if the write is a small asynchronous one, the request's response time can still be long. When a `sync` command is issued, the entire file is flushed to the disk to physically produce a new file.

We also collect the page cache consumptions after each test to understand memory efficiency of each system. Figure 7 shows the memory consumptions after the 2 MB overwrite tests. The cache consumptions of TotalCOW and Btrfs do not increase even for large files. It is because the page

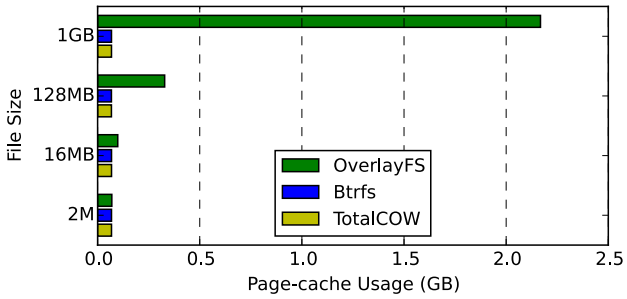


Figure 7: Page cache consumption after overwriting 2 MB data

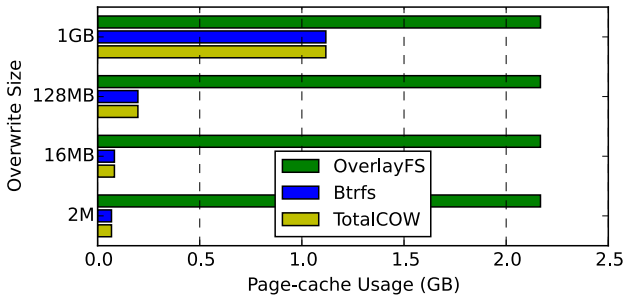


Figure 8: Page cache consumption after writing on a 1 GB file

cache only holds the overwritten 2 MB data. TotalCOW does not consume more memory than the stock Btrfs, because it does not allocate space in the new layer of cache for overwritten data. OverlayFS consumes significantly more cache space when overwriting a large file. It takes 2 GB in the page cache after writing only 2 MB data in a 1 GB file. OverlayFS leaves two files in the page cache after the file copying. One is the original file which supplies the data, and the other is a new file which receives the data.

Figure 8 shows memory consumptions of the overwrite tests on 1 GB files. Similar to the previous experiment, TotalCOW and Btrfs also efficiently use the page cache without apparent memory wastage. In contrast, OverlayFS consumes twice as much memory as the template file regardless the write size. It implies that the first-time write cost in OverlayFS is determined by the size of the template file, rather than the actual write size.

3.3 A Summary

As revealed in the performance results and analysis, both Btrfs and OverlayFS exhibit their respective advantages and disadvantages. On one hand, OverlayFS performs very well for sharing read-only files in the memory, while Btrfs incurs extra I/O load and consumes extra memory space. On the other hand, Btrfs performs ideally when doing copy-on-write, while OverlayFS shows inconsistent write performance and suboptimal page cache utilization with partial file modification.

TotalCOW addresses both of their disadvantages by extending the well-supported on-disk COW into in-memory cache space. Our design and experiments show that the all three objectives listed in Section 1 are achieved. By exposing key knowledge of on-disk sharing into the kernel space, cache memory and I/O resource can be efficiently used.

4. Related Works

Thin provisioning has been a major goal of the virtualization technology for various types of resources including memory and storage.

Virtual machines usually allocate minimal memory for guest OSs using delayed mapping [4]. The memory provisioning can also be reduced online by deallocating unused pages via balloon device [1]. Memory usage can also be reduced via compressing [14, 21] and deduplication [3, 13]. These techniques were proposed without exploiting knowledge on COW'ed files in the file system. In the context of page cache sharing, TotalCOW greatly improves the deduplication efficiency by reducing scope of search for redundant pages in the memory with the use of small amount of hint.

Virtualization systems employ memory deduplication techniques, such as Linux KSM (Kernel Same-page Merging) [3] and VMware's Transparent Page Sharing [13], to merge redundant pages across multiple virtual machines. Both methods use a scan approach to search for candidate pages for merging. Even though hashing or tree structures are used to accelerate the scan process, the overhead of memory deduplication can still be substantial. This is because the duplicated pages can be scattered across a large virtual memory space. Blindly scanning many non-duplicated pages is unnecessary and wastes CPU cycles. To avoid competing CPU cycles with other processes, the scan process has to be throttled [3], However, this throttling can introduce a long wait period for redundant pages to be detected and removed. These limitations make the current memory deduplication techniques less ideal for lightweight containers. TotalCOW is a hint-based approach to minimize the effort for searching redundant blocks. Furthermore, it does not place redundant pages into the page cache in the first place, while KSM attempts to remove duplicate pages that have existed in the memory.

Efforts on storage systems have been made to reduce the provisioning of the persistent storage media. For example, some file systems support deduplication and compression for their on-disk data [8, 10, 15, 16]. Due to the fact that these methods can lead to considerable computation cost, off-line approaches usually achieve better performance by employing more intelligent techniques for identifying redundant blocks [22]. Instead of directly reducing data size, copy-on-write technique saves storage space by preventing data growth while maintaining the logical correctness [11]. However, little attention has been paid on retaining the efficiency

when data reach the memory [12]. TotalCOW extends the space efficiency available in a COW file system to the memory for better utilization of I/O and cache resources. It is a complementary technique for file systems to providing better service to operating system and end users.

5. Conclusions

In this paper we propose two techniques and their implementation, collectively called TotalCOW, to address the performance and memory efficiency issue in COW-based file systems. TotalCOW uses a non-disruptive approach in VFS to enable in-memory COW. It complements the on-storage COW capability provided by the COW file systems. Compared to OverlayFS and UnionFS, TotalCOW retains the advantage of COW file systems. Experiments show that TotalCOW works efficiently for both read and write requests in terms of program execution time and memory efficiency.

6. Acknowledgments

We are grateful to the paper's shepherd Dr. Taesoo Kim and anonymous reviewers who helped to improve the paper's quality. This work was supported by US National Science Foundation under CAREER CCF 0845711 and CNS 1217948.

References

[1] Automatic ballooning. `http://www.linux-kvm.org/page/Projects/auto-ballooning`.

[2] Docker. `https://www.docker.com/`.

[3] Kernel samepage merging. `http://www.linux-kvm.org/page/KSM`.

[4] Kvm memory. `http://www.linux-kvm.org/page/Memory`.

[5] Linux containers. `https://linuxcontainers.org/`.

[6] Overlay filesystem. `http://goo.gl/VlBg98`.

[7] Why the world's largest hadoop installation may soon become the norm. `http://goo.gl/luPsxK`.

[8] Btrfs compression. `https://btrfs.wiki.kernel.org/index.php/Compression`, 2014.

[9] Microsoft unveils new container technologies for the next generation cloud. `http://goo.gl/OeNw1T`, 2015.

[10] J. Bonwick. `https://blogs.oracle.com/bonwick/entry/zfs_dedup`, 2009.

[11] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, 2003.

[12] J. Eder. Comprehensive overview of storage scalability in docker. `http://goo.gl/Hpe565`, 2014.

[13] F. Guo. Understanding memory resource management in vmware vsphere 5.0. `http://goo.gl/nSIzxG`, 2011.

[14] S. Jennings. Transparent memory compression in linux. `http://goo.gl/xAqSsP`, 2013.

[15] S. Jones. Online de-duplication in a log-structured file system for primary storage. Technical Report UCSC-SSRC-11-03, University of California, Santa Cruz, May 2011.

[16] X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace. Migratory compression: Coarse-grained data reordering to improve compressibility. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 257–271, Berkeley, CA, USA, 2014. USENIX Association.

[17] W. Mauerer. *Professional Linux Kernel Architecture*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[18] P. Mochel. The sysfs filesystem. In *Linux Symposium*, page 313, 2005.

[19] D. Quigley, J. Sipek, C. P. Wright, and E. Zadok. Unionfs: User-and community-oriented development of a unification filesystem. In *Proceedings of the 2006 Linux Symposium*, volume 2, pages 349–362, 2006.

[20] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[21] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 8–8, Berkeley, CA, USA, 1999. USENIX Association.

[22] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.