

File Systems: More Cooperations - Less Integration.

Alex Depoutovitch
VMWare, Inc.
aldep@vmware.com

Andrei Warkentin
VMWare, Inc.
andreiw@vmware.com

Abstract

Conventionally, file systems manage storage space available to user programs and provide it through the file interface. Information about the physical location of used and unused space is hidden from users. This makes the file system free space unavailable to other storage stack kernel components due to performance or layering violation reasons. This forces file systems architects to integrate additional functionality, like snapshotting and volume management, inside file systems increasing their complexity.

We propose a simple and easy-to-implement file system interface that allows different software components to efficiently share free storage space with a file system at a block level. We demonstrate the benefits of the new interface by optimizing an existing volume manager to store snapshot data in the file system free space, instead of requiring the space to be reserved in advance, which would make it unavailable for other uses.

1 Introduction

A typical operating system storage stack consists of functional elements that provide many important features: volume management, snapshots, high availability, data redundancy, file management and more. The traditional UNIX approach is to separate each piece of functionality into its own kernel component and to stack these components on top of one another (Fig. 1). Communication typically occurs only between the adjacent components using a linear block device abstraction.

This design creates several problems. First, users have to know at installation time how much space must be reserved for the file system and for each of the kernel components. This reservation cannot be easily changed after the initial configuration. For example, when deploying a file system on top of an LVM volume manager, users have to know in advance if they intend to

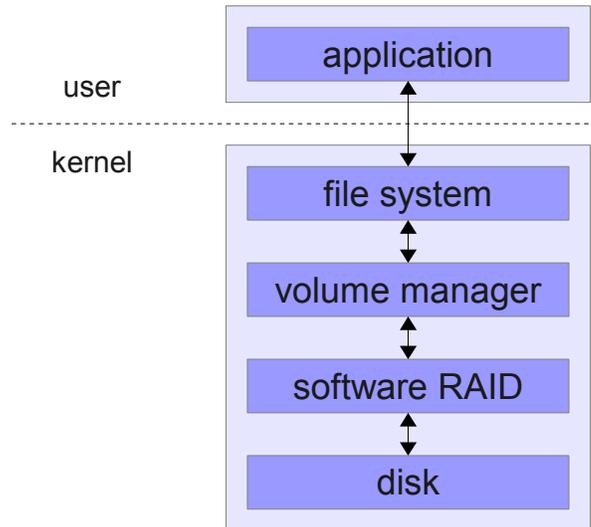


Figure 1: Storage stack

use snapshots, and they must choose the size of the file system so as to leave enough storage space to hold the snapshots [2]. This reserved space is not available to user programs and snapshots cannot grow larger than the reserved space. The end result is wasted storage capacity and unnecessary over-provisioning. Second, kernel components underneath the file system have to resort to storing their data within blocks at fixed locations. For example, Linux MD software RAID expects its internal metadata in the last blocks of the block device [4]. Because of this, MD cannot be installed without relocating an existing file system ¹.

To solve the above problems, many modern file systems, like ZFS, BTRFS, and WAFL, tend to incorporate a wide range of additional functionality, such as software RAID arrays, a volume manager, and/or snapshots [1, 6, 11]. Integrating all these features in a single software component brings a lot of advantages, such as flexibility and ease of storage configuration, shared free space pool, and potentially more robust storage al-

¹There is an option to store the metadata on a separate file system, but this requires a separate file system to be available.

location strategies. However, additional functionality of such “monolithic” file systems results in a complex and large source code. For example, BTRFS contains twice as many lines of code as EXT4, and four times more than EXT3. ZFS source code is even larger. A large code-base is more error-prone and more difficult to modify. BTRFS development started in 2008, and the stable version is yet to be released. In addition, these file systems put users into an “all or nothing” position: if one needs BTRFS volume management or snapshotting features, one has to embrace its slower performance in many benchmarks [13].

We believe that a file system can and should provide centralized storage space management for stateful components of the storage stack without integrating them inside the file system code. File systems are ideally suited for this role as they already implement disk space management. We propose a new interface, called *Block Register* (BR), to a file system that allows both user-mode applications and kernel components to dynamically reserve storage from a single shared common pool of free blocks. With BR interface, stateful storage stack components do not need to reserve storage space outside the file system, but can ask the file system to reserve the necessary space for them.

With the help of the new interface, file systems (such as EXT4) can benefit from more effective storage resource utilization, reduced wasted space, and flexible system configuration, without integrating additional functionality inside the file system. The BR interface also solves the problem of fixed metadata location, allowing kernel components to dynamically query for the location of necessary blocks. We demonstrate the benefits of the BR interface by integrating it with LVM snapshot mechanism and allowing snapshots to use file system free blocks instead of pre-allocated storage space. It is important that our interface can be exposed by existing file systems with no changes to their disk layout and minimal or no changes to their code.

2 Block Register Interface

The main purpose of the Block Register interface is to allow kernel components to ask the file system to reserve blocks of storage, increase and decrease number of reserved blocks, query for blocks that have been reserved before, and release blocks back for future reuse. While

brreserve(name, block_num)

brquery(name, offset, block_num) returns set of blocks

brexpand(name, block_num)

brtruncate(name, block_num)

brfree(name)

Table 1: Block Register interface functions.

we want this interface to provide both fine-grained control and extensive functionality, we also make sure that it can be implemented by any general-purpose file system without core changes to its design or its on-disk layout, and with no or minimal code changes. The interface needs to be generic and easy to use with simple abstractions. Although we think that kernel components will be the primary users of the Block Register interface, user mode applications might benefit from it as well, for example, to get access to data saved by kernel components. Therefore, the Block Register interface has to be accessible from both user and kernel modes.

The main abstraction we provide is a *reservation*. A reservation is a set of blocks on a block device, identified by their block numbers. These blocks belong to the caller and can be accessed directly by calling a block device driver for the device on which the file system is located. The file system will neither try to use these blocks itself nor include them in other reservation requests. Each reservation can be uniquely identified by its name. In the Table 1, we list functions of the Block Register API. Currently, we define 5 functions. When a caller needs to reserve a set of blocks, it calls the `brreserve()` function, specifying a unique name for this reservation and the number of blocks requested. In response, the file system reserves the required the number of blocks from its pool of free blocks.

When a caller needs to get the list of blocks of an existing reservation, it calls the `brquery()` function, passing it the name of the existing reservation and the range of blocks required. The range is specified by the offset of the first block in the query and the number of blocks to return. An existing reservation can grow and shrink dynamically by calls to `brexpand()` and `brtruncate()`. `brexpand()` takes the reservation name and the number of blocks to expand by, while `brtruncate()` takes the physical block to return back to the file system.

Finally, an existing reservation can be removed by calling the `brfree()` function. After a reservation is removed, all blocks belonging to the reservation are returned to the file system's pool of free blocks.

3 Linux Implementation

In this section, we will explore how the Block Register interface could be implemented in the EXT4 file system. Although we use some EXT4- and Linux-specific features, they are used only as a shortcut in the prototype implementation of the BR interface.

Although this is not necessary, we decided to represent Block Register reservations as regular files, using a common name space for files and reservations. Thus, `brreserve()` and `brfree()` can be mapped to file creation and deletion. Leveraging the existing file name space has other advantages with respect to access and management. Since reservations are basically normal files, they can be easily accessed with existing file system utilities and interfaces. Because the BR interface is oriented towards block I/O, application data access through the file system cache results in a data coherence problem and, therefore, should be prohibited. While open, reservation files must be also protected from modifications of file block layout, such as truncation. On Linux, this can be solved by marking the file as immutable with the `S_IMMUTABLE` flag.

Having the block reservations being treated as files has an additional implication. Invoking `brquery()` for every access would have a significant performance impact. Because of this, the caller of the BR interface may want to cache the results of previous `brquery()` calls. Thus, the file system cannot perform any optimization on the file, such as defragmentation, copy-on-write, and deduplication. To enforce this requirement, we relied on the Linux `S_SWAPFILE` inode flag, used to mark a file as unmovable.

We implemented `brexpand()` and `brtruncate()` using the `fallocate()` function implemented in Linux and EXT4 which allows changing the file size without performing actual writes to the file. There are, however, a few specifics of `fallocate()` behaviour that have to be taken into consideration. First, `fallocate()` makes no guarantees on the physical contiguity of allocated space. This may affect I/O performance. Second,

`fallocate()` call results in write operations to the underlying block device, thus special care has to be taken by the caller in order to avoid deadlocks.

`brquery()` was implemented using the `bmap()` function, which returns the physical device block for the given logical file block. `bmap()` may also trigger some read requests to the underlying storage.

4 Snapshots with Block Register Interface

In order to evaluate the Block Register interface, we modified Linux Logical Volume Manager (*LVM*) to use our new interface to allocate storage necessary to create and maintain snapshots of block devices [2]. In this section, we briefly describe the changes we made to the snapshotting code so that it can make use of the Block Register interface.

Snapshots preserve the state of a block device at a particular point in time. They are widely used for many different purposes, such as backups, data-recovery, or sandboxing. LVM snapshots are implemented using device mapper. Snapshots require additional storage to hold the original version of the modified data. The size of the additional storage depends on the amount of data modified during the lifetime of the snapshot. Currently, users planning to use snapshots create a file system only on a part of available storage, reserving the rest of it for snapshot data. The space reserved for snapshots cannot be used to store files, and its size is fixed after the file system creation.

We modified `dm-snap`, the device mapper target responsible for creating snapshots, to use the Block Register interface to allocate space to store the snapshot data as needed instead of using space reserved in advance. `dm-snap` uses this space to maintain the set of records, called an exception store, that describe chunks of the original device that have been modified and copied to the snapshot device as well as the old contents of these chunks. We created a new exception store persistence mechanism which uses block reservations instead of a block device for storing data. We based our code on the existing exception store code, `dm-snap-persistent`, and reused its metadata layout.

In order to create a snapshot of a file system that supports the Block Register interface, the user specifies the device with the file system and the name to be used for

a block reservation on that file system. This reservation will contain the snapshot data. The new dm-snap code calls `brreserve()` for the initial block reservation. Responding to this request, the file system creates a new reservation. dm-snap queries blocks allocated to this reservation and creates a new snapshot using the reserved blocks.

After a certain amount of changes to the file system, dm-snap will need additional blocks to maintain the original version of the file system's data. In order to get these blocks, dm-snap calls `brexpand()` to request the new set of blocks. Because expanding the reservation might cause additional changes to the file system metadata, this call cannot be made in the context of a file system operation, otherwise, deadlock might occur. In order to avoid the deadlock, the new dm-snap maintains a pool of available free blocks. If the pool falls below a "low watermark", a separate thread wakes up and requests additional blocks through the Block Register interface. The value of the low watermark depends on the write throughput of the file system.

To improve performance and prevent callbacks to the file system in the context of an I/O operation, dm-snap caches the data returned by `brquery()` in memory and queries this data with a binary search combined with a most-recently-used lookup. The implementation of the cache mechanism has been largely based on the obsolete dm-loop prototype [3].

During the operating system boot process, before the file system can be mounted in read-write mode, it has to be mounted in read-only mode, so LVM can query the file system for blocks reserved for snapshot data. After that, LVM enables the copy-on-write mechanism for the block device containing the file system and exposes the snapshots to the user. Once the COW mechanism is enabled, the file system can be remounted in read-write mode.

5 Evaluation

In order to evaluate our changes, we incorporated them into the Linux 3.2.1 kernel and measured their impact on performance of the file system in the presence of a snapshot. We compared our snapshot implementation using the Block Register interface and the standard LVM snapshot implementation. The results are presented in Figure 2. Each measurement shows the performance of

the file system in the corresponding benchmark with an active snapshot implemented using the Block Register interface relative to performance of the same benchmark with a snapshot taken using the standard Linux LVM implementation. Each result is the average of 3 runs. Values greater than 100% mean that the Block Register implementation outperforms the standard Linux implementation.

In the first set of experiments, we created a 50GB test file on the file system, took a snapshot of the file system, and started issuing I/O requests to the file on the original file system. During each experiment we maintained the number of outstanding I/O operations (*OIO*) to be constant (either 1 or 8) and measured the number of completed I/O operations per second. Each I/O operation had a size of 4096 bytes. We used direct synchronous I/O to avoid caching effects. Since read operations do not involve snapshotting, we did not notice any difference between our and the standard Linux snapshot implementations. However, our snapshot implementation behaved better while serving write operations. In random write operations, performance improvement varies from 1% for *OIO*=1 to 8% for *OIO*=8. Performance gain for sequential write operations is more significant: 9% for *OIO*=1 and 26% for *OIO*=8.

In the second set of experiments, we created a snapshot of an empty file system and measured the time necessary to unpack 5 copies of the Linux 3.2.1 kernel source tree from an archive to the file system and then delete all created files. The last column on Figure 2 shows the result of this experiment. Our implementation performs on a par with the standard Linux implementation in this benchmark.

We believe that the performance improvement comes from a shorter disk head seek span during write operations with our snapshot implementation. As shown on Figure 3, copy-on-write operation for a block requires two additional seek operations. The first is to seek to the location of the block B2 in the snapshot to write the original data of the block B1, and the second is to seek back to the original block B1 to write the new data. In the standard Linux snapshot implementation, storage blocks containing snapshot data are located in the reserved area outside the file system. This forces the disk to perform seek operations on average over the span of the whole file system. In our implementation, a file system has a chance of placing these two blocks closer to each other, thus reducing the seek time. Initially, when

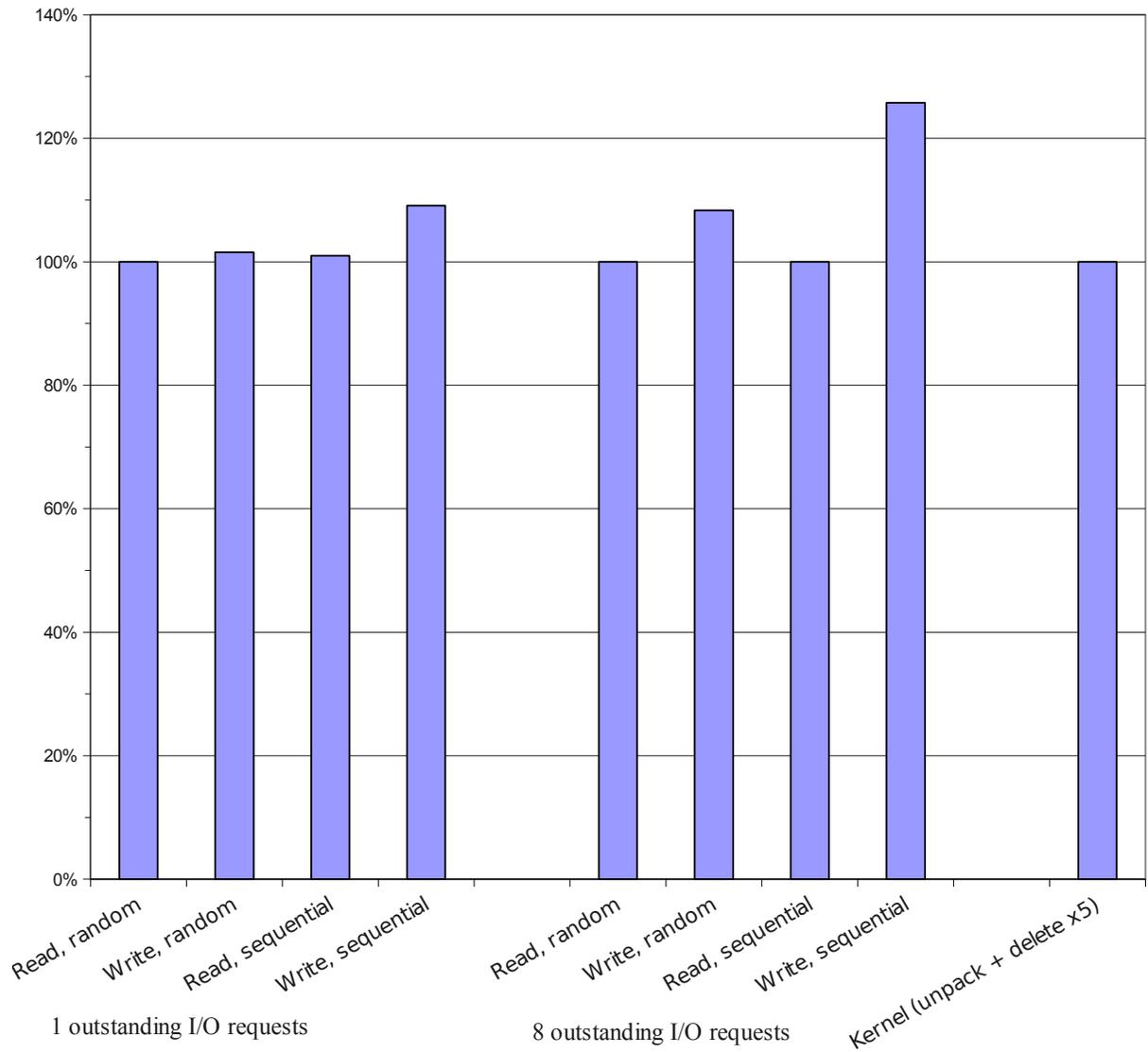


Figure 2: Performance of snapshots using Block Register relative to LVM

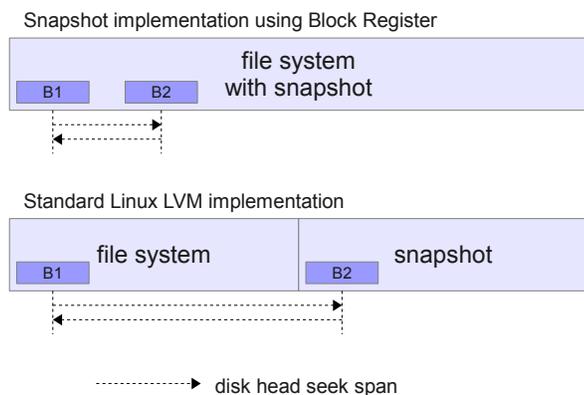


Figure 3: Disk head seek span during copy-on-write operation

we designed the Block Register interface, we did not target performance improvements. This result comes as a pleasant side effect and leaves room for future work on snapshot placement optimizations.

6 Open Issues

The implementation of the Block Register interface raises several important problems. The best way to solve them is yet to be identified. In this section, we describe some of these problems.

So far, we considered only file systems located on a single partition, disk, or a hardware RAID array. Currently, BR interface will not work in case of an additional component, which performs non-linear mapping of disk blocks, placed between the file system and the user of the BR interface. An example of such a component is a software implementation of RAID 0.

Block reservation and query may result in additional I/O operations. Therefore, special care has to be taken, e.g. calling them asynchronously, to avoid deadlock when calling these functions while serving other I/O requests. Another solution is to mark I/O caused by reservation requests with special hints, so that kernel components can give them priority and guarantee that they will not be blocked.

Blocks that belong to reservations must be protected from some file system operations, such as defragmentation or accidental modifications by user-mode applications.

The block allocation algorithm depends on the file system and, therefore, may be sub-optimal in some cases because file systems are tuned for the most common case.

Finally, there are some implementation limitations, such as caching reservation block numbers in memory, which may become a problem for very large and fragmented reservations.

7 Future Work

We are going to investigate additions to the BR interface that will allow other components of the storage stack to interact with a file system without being incorporated into it. The goal of this interaction is to achieve flexibility and functionality similar to those provided by “monolithic” file systems, like BTRFS or ZFS.

Other kernel block layer components may benefit from the Block Register interface. We are planning to modify MD software RAID and DRBD, the Distributed Reliable Block Device [8], to use the Block Register interface for storing their metadata within the file system.

Another interesting opportunity comes from our implementation of reservations as regular files in a file system. Because of that, they could be accessed (with necessary precautions, such as taking measures to avoid the cache coherence problem) by user mode-programs. This allows, for example, to copy or archive contents of the file system along with snapshots using familiar utilities, like *cp* or *tar*.

We are also going to investigate changes to block allocation algorithms that allow kernel components to request reservations in the proximity of a specific location. This can improve performance of some operations, like copy-on-write or update of dirty block bitmap, by enabling proximal I/O [14].

8 Related Work

Some file systems, such as BTRFS, ZFS or WAFL, implement snapshots internally and also store original version of the modified data in their own free space [1, 6, 11]. Our approach uses a similar technique, however, it does not depend on a particular file system implementation.

Thin provisioning tries to achieve similar goals, however, once a block has been written to, a thinly provisioned block device does not have knowledge if this block contains useful data or not. In order to solve this problem, a file system has to support additional interfaces to the block layer, telling the block layer when blocks become free. Another problem with thin provisioning is in the duplication of file system functionality for accounting and allocating storage blocks. This results in additional levels of indirection and additional I/O for storing persistent allocation information. Conflicting allocation policies may result in less than optimal performance, e.g. blocks that file system allocator tries to allocate continuously might be allocated far from each other by a thin provisioning code. Examples of thinly provisioned block device implementations include Virtual Allocation and dm-thin [5, 12].

Linux provides *fibmap* and *fiemap* ioctl calls, which return information about the physical location of files on the block device to applications [9]. Wires et al. argue that growing number of applications can benefit from the knowledge of the physical location of file data [15]. They present a MapFS interface that allows applications to examine and modify file system mappings between individual files and underlying block devices. Block Register, on the other hand, provides applications and kernel components access to the free space of the file system. Block I/O to files using `bmap()` has been used in Linux to implement a file-backed swap device [7].

Parallel NFS (*pNFS*) is an enhancement of the NFS file system that exposes file layout to the clients in order to improve scalability [10]. While pNFS work is concentrated on large-scale networks, we argue that similar ideas can be successfully applied in a single-machine environment.

9 Conclusion

In this paper, we proposed a novel interface that extends usefulness of the file systems. Our interface is generic and does not rely on a specific file system implementation. It allows kernel storage stack components to share storage space with user files and use the file system to reserve storage space and locate data. On the example of storage snapshotting, we showed how our interface can be used for more flexible and efficient storage utilization without integrating snapshots inside file system's code. In addition, we demonstrate that the new interface may also help optimize storage I/O performance.

10 Acknowledgments

The authors express sincere gratitude to Edward Goggin for his insightful advice regarding this work.

References

- [1] BTRFS project. <http://btrfs.wiki.kernel.org>.
- [2] Device-mapper resource page. <http://sources.redhat.com/dm/>.
- [3] dm-loop: Device-mapper loopback target. <http://sources.redhat.com/lvm2/wiki/DMLoop>.
- [4] Linux software RAID. <http://linux.die.net/man/4/md>.
- [5] Thin provisioning. <http://lwn.net/Articles/465740/>.
- [6] 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.
- [7] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 3rd edition, 2006.
- [8] L. Ellenberg. Drbd 9 and device-mapper: Linux block level storage replication. *Proc. of the 15th International Linux System Technology Conference*, pages 125–130, 2008.
- [9] M. Fasheh. Fiemap, an extent mapping ioctl. <http://lwn.net/Articles/297696/>.
- [10] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. In *Proc. of the 22nd IEEE Conference on Mass Storage Systems and Technologies*, pages 18–27. IEEE, 2005.
- [11] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an NFS file server appliance. *Proc. of the Winter USENIX Conference*, pages 235–246, 1994.
- [12] S. Kang and AL Reddy. An approach to virtual allocation in storage systems. *ACM Transactions on Storage*, 2(4):371 – 399, 2006.

- [13] Michael Larabel. Linux 3.3 kernel: BTRFS vs. EXT4. <http://www.phoronix.com>, 2012.
- [14] J. Schindler, S. Shete, and K.A. Smith. Improving throughput for small disk requests with proximal I/O. In *Proc. of the 9th USENIX Conference on File and Storage Technologies*, 2011.
- [15] J. Wires, M. Spear, and A. Warfield. Exposing file system mappings with MapFS. In *Proc. of the 3rd USENIX conference on Hot topics in storage and file systems*, 2011.