Exposing File System Mappings with MapFS

Jake Wires, Mark Spear, Andrew Warfield
University of British Columbia
{*jtwires,mspear,andy*}*@cs.ubc.ca*

Abstract

The conventional model of a file as a contiguous array of bytes hides information about the physical location of data from users. While this simplifying abstraction can be useful in some cases, it can also lead to suboptimal performance and unnecessary overhead. A growing number of applications – even those as basic as the Unix *cp* utility – can benefit from increased access to file system metadata. We present MapFS, a file system which allows applications to create, inspect, modify, and remove the mappings established between individual files and physical storage. MapFS gives users increased power and flexibility, facilitates true end-to-end application design, and optimizes many common file system tasks.

1 Introduction

The POSIX API provides a lowest-common-denominator interface for working with file system objects. While this interface has not changed for decades, both the file systems which implement it and the applications which consume it continue to grow in complexity and sophistication. For many modern applications, the POSIX API presents more of a barrier than a useful interface. This is particularly evident for a growing class of software which is expressly concerned with issues like on-disk data placement. Rather than making this information visible in a meaningful way, the POSIX API intentionally hides it. We therefore present MapFS, a file system which exposes many powerful features of existing file systems via a novel API.

MapFS allows applications to create, inspect, modify, and remove the mappings established between individual files and physical storage. MapFS's extended interface is compatible with the standard POSIX interface, but it gives end users greater knowledge and control of data placement. By exposing the logical-to-physical mapping behind file objects, MapFS allows applications to exploit domain-specific knowledge to arrange data in optimal fashion rather than relying on the assumptions of file system developers.

It also facilitates the very common task of shuffling data between files. A plethora of applications, ranging from basic Unix utilities like *tar* to sophisticated video editing software, spend a lot of time and effort moving data blocks from one file to another. Mundane as this task is, current file system interfaces do not provide an elegant way to accomplish it. The naive approach of using the POSIX `read` and `write` procedures is well-documented for its inefficiency [9]. Specialized Linux system calls like `sendfile`, `tee`, and `splice` offer improvements when copying blocks to sockets or pipes, but do not help when copying data from one file to another. A clever use of `mmap` can eliminate some of the page cache overhead of file-to-file copies, but it misses an obvious point: modern file systems like ZFS [3] and Btrfs [7] can perform this task in O(1) time simply by manipulating file-to-disk mappings. What is actually needed, and what MapFS provides, is the storage analogue of `mmap`: an interface for controlling the mappings between files and disk blocks.

Of course, MapFS offers more than just an optimized copy routine. The MapFS interface can be used to implement a variety of useful features, including deduplication, in-place deletion of file ranges, sub-file copy-on-write policies, rapid file aggregation, etc. The bottom line is that many applications today can benefit significantly from having greater control over file system mappings. And the copy-on-write functionality already available in newer file systems makes it relatively easy to extend this control to users, because most of the requisite machinery is already in place. By exposing this mechanism to userspace applications via a clean API, MapFS dramatically improves the performance of a number of common file system tasks and enables novel implementations of new features.

2 Design and Implementation

A primary goal of MapFS is to give users greater control of file mappings without exposing excessive file system gore. File systems are complex, mission-critical pieces of software which already support a fairly wide interface. They have many moving parts and require sophisticated data structures to manage file mappings. While we want to give users greater control of these mappings, we do not want to undermine file system reliability or push excessive complexity into application logic – for example, *cp* should not have to understand versioned b-trees to manipulate mappings. In this section we propose an initial MapFS API and describe its implementation.

2.1 MapFS API

The interface exported by MapFS must strike a balance between offering sufficient flexibility and providing a clean abstraction. And since we fully expect many new and unforeseen uses of MapFS as applications evolve to take advantage of the new functionality, we want a reasonably generic interface.

Table 1 presents our working API for MapFS. We currently provide three new file system operations: `fmap`, `fremap`, and `fsplice`.

`fmap` returns a structure describing the logical-to-physical mapping of a file range in an extent-based format. A number of file systems already offer similar functionality via the `fiemap` ioctl.

`fremap` allows a region of one file to be mapped into a region of another file. It takes a source file and offset, a destination file and offset, and size and type parameters. The type parameter allows applications to specify which type of mapping should be created. At the moment we only support copy-on-write mappings, but we are investigating the potential for other types, such as shared-write mappings. Existing mappings are split or removed altogether to accommodate new mappings. Note that the source file can be a block device, allowing applications to control block allocation policies.

`fsplice` allows the insertion and removal of logical address ranges in a given file. This function enables applications to insert and delete data at arbitrary file offsets without having to shift the physical location of subsequent data. Newly inserted address ranges are not initially mapped to disk; reads to these ranges return all zeros until they are written or remapped. `fsplice` deletions deallocate any physical blocks mapped to the given logical addresses and shift subsequent mappings to fill the gap. This is somewhat similar to using `ftruncate` to reduce the size of a file, except `fsplice` calls are not restricted to operating from the end of a file – they can be applied to arbitrary file ranges.

fmap(*fd*, *off*, *size*) returns *map*
fremap(*sfd*, *soff*, *dfd*, *doff*, *size*, *type*)
fsplice(*fd*, *off*, *size*, *insert/remove*)

Table 1: **The MapFS API**

2.2 Implementation

The fundamental concept behind MapFS does not require a radical new file system design or implementation. Admittedly, older file systems like ext3 and NTFS would require significant restructuring to support the MapFS API. But newer file systems like ZFS and Btrfs can support it quite naturally, for a number of reasons:

- they are extent-based, allowing for an efficient description of arbitrary byte-granularity mappings;
- they support copy-on-write, making it easy to share data and still provide isolation across mappings;
- they already implement much of the bookkeeping required to support the MapFS API.

We have built a prototype implementation of MapFS based on Btrfs. We make use of an existing Btrfs ioctl, `BTRFS_IOC_CLONE_RANGE`, to perform much of the hard work of creating new mappings. MapFS adds around 200 lines of code to the Btrfs sources and presently supports only the `fremap` function. Mappings are currently restricted to page granularity, but we plan to support byte-granularity mappings as our implementation matures. Applications can still benefit from MapFS in spite of this limitation by falling back to traditional `read`/`write` solutions for non-aligned portions of map regions.

3 Challenges

Even though MapFS is a natural extension of Btrfs, there are a number of technically challenging issues to overcome before the API can be fully supported. Chief among these is the difficulty of allowing byte-granularity mappings, which can wreak havoc on page alignment. Other significant challenges include addressing the performance implications of mapped files and ensuring cache coherency in the face of arbitrary `fsplice` operations.

3.1 Alignment

Figure 1 illustrates how arbitrary byte-granularity mappings can produce files that are poorly aligned on disk. File *f2* in the figure could have been created with the following invocation:

`fremap(`*f1*`, 1, `*f2*`, 0, `*f1*`.length, CoW)`

This would leave file *f2* misaligned on disk, which has a number of implementation and performance implications.

First, while Btrfs extents are well-suited for describing misaligned ranges, the current implementation is not – there are various places throughout the code where proper 4K alignment is assumed. This has lead to a few bugs when dealing with misaligned mappings, but resolving these issues is a small matter of programming.

More importantly, however, is the fact that bounce buffers will likely be required to read and write misaligned regions. This complicates the file system implementation and has negative effects on both performance and resource consumption.

Given these difficulties, it is reasonable to ask whether MapFS should support byte-granularity mappings at all. When storage consumption is a bigger concern than IO performance, misaligned mappings might be justified. But it may prove wiser to force proper alignment on calls to `fremap`, as is already done for `read` and `write` calls against files opened with O_DIRECT.

To better understand the tradeoffs involved here, we plan to implement byte-granularity mappings and evaluate the consequent overhead.

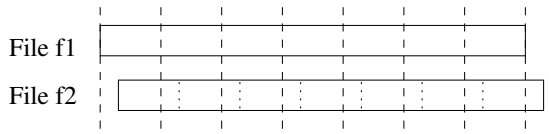

Figure 1: **Alignment issues with MapFS** *Byte-granularity* `fremap` *calls can lead to misaligned files; in this case, file f2 has been remapped in such a way that it does not begin at a page boundary on disk.*

3.2 Performance Implications

While MapFS makes tasks like file aggregation almost free, it can lead to fragmentation and reduced throughput. Aggregated files, for example, will not be contiguous on disk. While this tradeoff is not unique to MapFS (in fact, it is inherent in any copy-on-write file system), it is worth considering what can be done to reduce fragmentation when possible.

For infrequently-accessed backup files, fragmentation is an easy price to pay in return for storage savings and rapid aggregation. On the other hand, when performance is a crucial concern, it may make sense to defragment such files. This could potentially be done automatically in the file system by tracking access patterns and rearranging the on-disk layout of data to optimize for the common case.

3.3 Open Issues

MapFS poses many questions about issues like concurrency, cache coherence, security, and portability. We list a few of them here:

- It is not currently clear how non-length-preserving operations in the middle of a file should affect users who have file pointers positioned beyond the point of insertion or deletion.
- MapFS operations will need to be synchronized against in-flight block IO requests. This is presently achieved with coarse-grain locking, but more efficient solutions may be feasible.
- MapFS must guarantee consistency in the face of arbitrary system crashes, so mapping updates will likely need to be atomic or journalled.
- Special care will need to be taken to ensure that the page cache remains coherent in the face of remappings and splice operations.
- The MapFS API could potentially introduce new attack vectors for illicitly manipulating file system data. We expect existing access control mechanisms like capabilities and file permission bits will provide adequate security.
- Legacy applications will need to be modified before they can benefit from MapFS. So far we have had little trouble porting applications to the new API, but finding ways to automate this process as much as possible would provide an obvious improvement.
- It is unlikely that all file systems will support the MapFS API, so applications will need fallback implementations to remain portable. This could be partially addressed by simulating MapFS functionality with standard POSIX API calls (much like `posix_fallocate` does), but it is not clear that such a generic approach is optimal.

4 Evaluation

As part of a preliminary evaluation of MapFS, we have ported two applications to use the new interface: *tar*, the common Unix archival utility, and *vhd-util*, a tool from the Xen [2] virtualization stack which manipulates VHD [8] virtual disk images.

4.1 *tar*

tar is a standard command line tool for creating archives. It defines a format for representing file system data and metadata in a single archive file, with full support for objects like directories and symlinks. *tar* archives are often retained as read-only backups of important file system data.

Files are added to *tar* archives by appending the file contents, along with metadata describing their names and attributes, to the archive file. In a conventional file system, this entails reading the files in their entirety and writing verbatim copies of them to the archive file.

We modified *tar* to use the MapFS interface to directly map data from target files into the archive file. This completely eliminates the need to read or write the target files. Copy-on-write references allow the archive file to reference the original data without requiring duplicate copies. This essentially gives users explicit, fine-grain control of the automatic, system-wide snapshots provided by file systems like WAFL.

Because MapFS currently only supports page-aligned mappings, not all files can be completely mapped into the archive. Misaligned portions of files are copied in the conventional manner. Our modified *tar* uses approximately 1% of the disk space (including file data and file system metadata) required for the standard *tar*.

We measured the time required to archive file system hierarchies of varying size populated by Impressions [1] using both the standard *tar* utility and our modified version. The results plotted in Figure 2 are the average of five runs for each configuration; standard variation is shown. As expected, the modified *tar* performs much better than its unmodified counterpart.

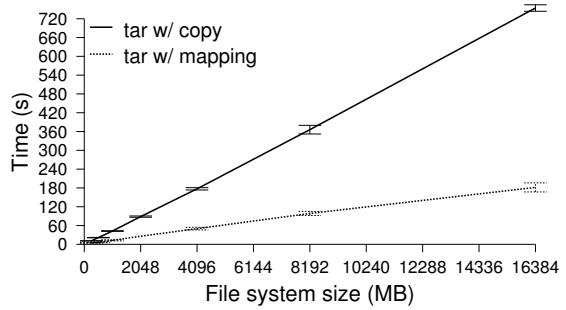

Figure 2: *tar* **Performance in MapFS**

4.2 *vhd-util*

Many virtualization systems use file formats like VMDK and VHD to represent virtual disks. These file formats support the notion of delta files, which are copy-on-write point-in-time snapshots of a virtual disk. As illustrated in Figure 3a, delta files are linked in a chain to their original root image. As more snapshots are created, this chain grows longer, and the overhead of traversing it increases. It is thus necessary to prune snapshots periodically. This is achieved by a process known as *coalescing*, whereby all the data from one delta file is copied onto to its parent. Once the copy is complete, the child delta file can be deleted.

VHD snapshots can grow quite large, and coalesce operations can often entail copying tens of GBs. We modified *vhd-util*, which is part of the Xen toolstack, to use the MapFS interface for coalescing VHDs. Rather than copying blocks from one VHD to another, the modified vhd-util maps them directly into the destination file.

We measured the time required to coalesce VHD snapshots containing file system images of varying sizes, again populated by Impressions. As Figure 4 shows, our modified version of *vhd-util* significantly outperforms the original.

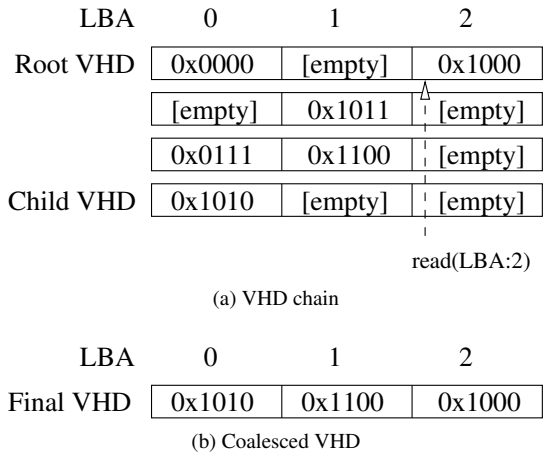

Figure 3: **VHD Virtual Disk Images** *Figure 3a depicts a VHD chain with 3 point-in-time snapshots. Reading from a VHD chain entails visiting each VHD in the chain until valid data is found; in this case, reading logical block 2 would require visiting all VHDs in the chain. Figure 3b shows the result of a coalesce operation on the VHD chain: relevant data from child snapshots are copied into the parent image and the children images are deleted.*

5 Future Work

We expect the MapFS interface will be useful for a variety of interesting applications. Deduplication [6], currently a hot topic in storage research, could be implemented for MapFS as a system-level service that runs as a background userspace task. MapFS also provides a natural interface for userspace defragmentation tools.

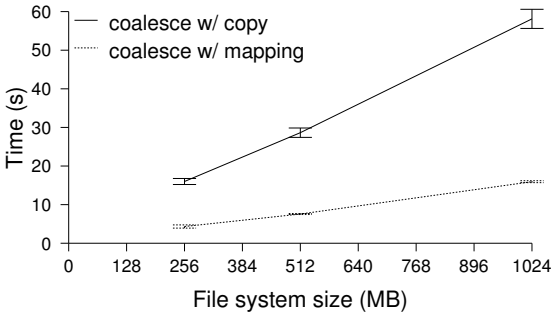

Figure 4: *vhd-util* **Performance in MapFS**

Going further, MapFS could support novel mapping relationships, such as 1-to-N mappings. Consider, for example, *ureadahead*, which strives to reduce Linux boot times by intelligently pre-loading the page cache with files used during system startup. Disk seeks contribute to much of the overhead *ureadahead* is trying to eliminate. With MapFS, *ureadahead* could map boot files into an on-disk lookaside cache, leading to faster, more efficient boots. This would represent a switch from logical-to-logical mappings (mappings from one file to another) to logical-to-physical mappings. We are currently exploring how best to expose this functionality.

Finally, an intriguing addition to MapFS would extend the mapping space to include targets other than disk blocks, allowing file regions to be mapped to sockets and even applications. One could imagine mapping a file to a computationally expensive application and using the page cache for memoization. Mappings could also be used to install userspace hooks to file system code paths, providing similar functionality to FUSE [10], the userspace file system.

6 Related Work

Some of the ideas behind MapFS have already found their way into the Linux kernel, emphasizing that the need to expose file system metadata to applications is a real-world problem. Basic support for exporting read-only access to file mappings has evolved from the early `fibmap` ioctl to the more recent `fiemap` [4] ioctl. These interfaces are intended to make the mappings between logical files and disk blocks visible to applications. `fiemap`, for example, has already been incorporated into the Linux utility *cp*, allowing it to avoid reading from holes when copying files, thus eliminating wasted cycles and reducing page cache churn. MapFS is a natural extension of these methods, allowing users to create and manipulate file mappings as well as read them.

MapFS is similar in spirit to the Exokernel [5], which strives to give end users as much control over resources as is safely possible. XN, the Exokernel's storage system, takes the radical approach of allowing applications to define arbitrary file system metadata formats. Metadata structures are formalized by application-provided *untrusted deterministic functions*, which the kernel uses to enforce protection. MapFS metatdata, on the other hand, is fixed in format and understood by the kernel. But by exposing this metadata in a controlled way through a narrow API, MapFS can extend significant power and flexibility to applications, granting them similar end-to-end benefits to those provided in Exokernel.

7 Conclusion

The assumption that applications do not care about data placement is clearly wrong, but it seems baked into many file systems. MapFS provides a novel interface which offers users increased control of their data. In so doing, it optimizes many common file system tasks and enables elegant implementations of new features.

References

[1] AGRAWAL, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Generating realistic impressions for file-system benchmarking. In *In Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST* (2009), pp. 125–138.

[2] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 164–177.

[3] BONWICK, J., AND MOORE, B. Zfs: The last word in file systems, 2007. http://www.sun.com/software/solaris/zfs$_l c_p reso.pdf$.

[4] FASHEH, M. Fiemap, an extent mapping ioctl, 2008. http://lwn.net/Articles/297696/.

[5] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICEO, H. M., HUNT, R., MAZIRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on exokernel systems. In *In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles* (1997), pp. 52–65.

[6] KULKARNI, P., DOUGLIS, F., LAVOIE, J., AND TRACEY, J. M. Redundancy elimination within large collections of files. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2004), ATEC '04, USENIX Association, pp. 5–5.

[7] MASON, C. btrfs wiki, 2011. http://btrfs.wiki.kernel.org.

[8] MICROSOFT. Virtual hard disk image format specification, 2009. http://technet.microsoft.com/en-us/virtualserver/bb676673.

[9] STANCEVIC, D. Zero copy i: User-mode perspective. *Linux Journal*, 105 (January 2003). http://www.linuxjournal.com/article/6345.

[10] SZEREDI, M. Fuse: File system in user space, 2011. http://fuse.sourceforge.net.