DataWarp Transparent Cache: Data Path Implementation

Matt Richerson
Cray Inc., Bloomington, MN 55425
Email: mattr@cray.com

***Abstract*—DataWarp transparent cache uses SSDs located on the high speed network to provide an implicit cache for the parallel filesystem. We will look at which components make up the data path for DataWarp transparent cache and see how they interact with each other. The implementation of each component is discussed in depth, and we will see how the design decisions affect performance for different I/O patterns.**

***Keywords*-Filesystems & I/O, DataWarp, SSD**

I. Introduction

DataWarp (DW) transparent cache uses SSDs on dedicated servers to provide a cache between the parallel filesystem (PFS) and compute nodes. The cache is implicitly managed by DataWarp, and the namespace seen on the compute nodes is identical to the PFS. I/O through DataWarp targets the SSDs and typically provides higher bandwidth and lower latency than using the PFS directly. Implicit caching makes it easy for existing applications to start using DataWarp since its interface is similar to that of the PFS.

DataWarp transparent cache is built on the existing DataWarp scratch data path. DataWarp scratch is an existing DataWarp product that allows explicit file level caching of the PFS onto an SSD backed filesystem. Users manually stage data between the scratch SSDs and PFS using batch job directives or a user library available at application runtime.

The goal of this paper is to provide implementation details on the DataWarp data path. We will take a brief look at each of the components to provide some background, and then we will see how they are put together to form the DataWarp scratch data path. Next we will see how that data path was expanded to add implicit caching functionality and discuss the challenges associated with it. Finally, we will look at the performance characteristics of DataWarp transparent cache to show what workloads it's useful for.

II. Components

DataWarp provides applications running on compute nodes access to fast SSD space on the DW servers. Most configurations will have multiple DW servers that each have local SSDs attached. The job of the DataWarp data path is to make these separate SSDs look like a single contiguous storage space to the application. This is done with a distributed filesystem that manages data across all the servers and provides a POSIX filesystem interface on the clients. The DataWarp team has developed its own distributed filesystem to make the best use of the DW hardware. This distributed filesystem is made up of multiple different components that are implemented as kernel modules. This section provides a brief overview of the each of the components in the data path.

A. wrapfs

Some of the DataWarp components are kernel level filesystems based on the GPL community wrapfs project. wrapfs is a stackable, pass-through filesystem which is layered between a traditional lower filesystem (e.g., ext4) and the kernel virtual filesystem (VFS). The goal of wrapfs is to insert a layer in the filesystem stack that has no effect. wrapfs registers VFS operations with the kernel VFS layer and passes them down to the lower filesystem without modification. The kernel VFS layer only interacts with wrapfs and not the lower filesystem.

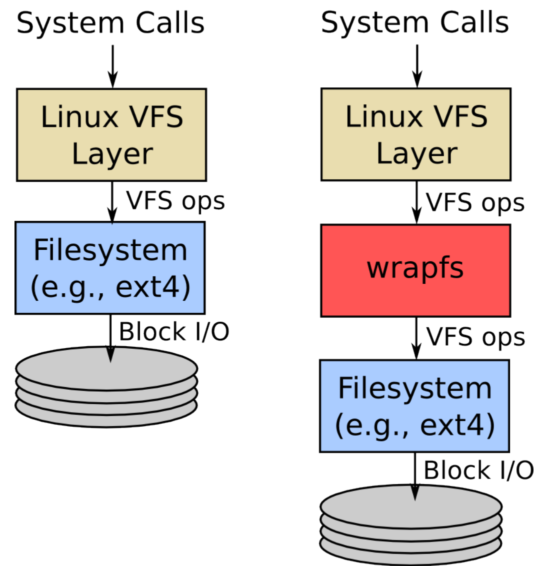

Figure 1. Traditional filesystem stack compared to a filesystem stack with wrapfs.

In order to remain completely separate from the lower filesystem, wrapfs maintains its own set of in-memory filesystem data structures (e.g., inodes) that are duplicates of the lower filesystem. wrapfs updates the data structures after each VFS operation to keep them consistent with the lower filesystem. Any direct access to the lower filesystem causes the wrapfs data structures to become stale. For

this reason, it is important that all accesses occur through wrapfs. Duplicating memory structures means that memory consumption is increased when using wrapfs, however, the performance impacts of this extra layer are minimal.

By itself, wrapfs does not provide any useful functionality since it presents exactly the same information as the lower filesystem. However, it serves as an excellent template for implementing new stackable filesystems. Later we will show how we expanded upon the wrapfs code base to create new filesystems for DataWarp.

B. Data Virtualization Service

The Datawarp data path uses the existing data virtualization service (DVS) component. DVS is a kernel level I/O forwarder which allows access to a filesystem mounted on a remote node.

DVS can be divided into client and server components. The client provides a user facing POSIX filesystem interface, and the server interacts with a lower filesystem. All VFS operations on the DVS client are forwarded over the network to the DVS server. Those operations are called on the lower filesystem, and DVS sends the results back to the client. Since DVS is forwarding all I/O operations, there is relatively little state it needs to store on the client and server. This allows DVS to scale up to a very large number of clients as no client-to-client communication is required.

A typical way that DVS is used is to project a parallel filesystem as the lower filesystem. In this configuration DVS can have multiple servers that are parallel filesystem clients. The parallel filesystem maintains coherency between the DVS servers, so DVS clients can forward I/O to any server. However, DVS typically chooses which server to target in a way that limits the amount of coherency work the lower filesystem has to perform. All DVS clients send metadata operations for a particular inode to a single server based on a hash of the inode number. Data operations for a single inode can be striped across multiple servers, but the range of the stripes do not overlap. This prevents lock thrashing between the parallel filesystem clients.

C. Data Virtualization Service IPC

DVS IPC is an inter-process communication layer that DVS uses to send network messages between the DVS client and server. The underlying transport that DVS IPC uses is LNet, Lustre's network layer.

D. DataWarp Filesystem

The DataWarp filesystem (kdwfs) is a Cray Inc. developed filesystem based on wrapfs. kdwfs runs on the DW servers and is responsible for handling the communication between servers that a distributed filesystem requires. It uses the DVS IPC layer to send messages over the network. kdwfs will be discussed in detail in the scratch data path section.

E. Data Caching Filesystem

The data caching filesystem (kdcfs) is another Cray Inc. developed filesystem based on wrapfs. kdcfs also resides on the DW servers, and it is used to manage data movement between the SSD and the PFS. The section on DataWarp transparent caching will give an in-depth implementation view of kdcfs.

F. XFS

XFS is a high performance local filesystem that is used on the DataWarp servers. XFS was chosen based on some of the features it provides which will be described in a later section.

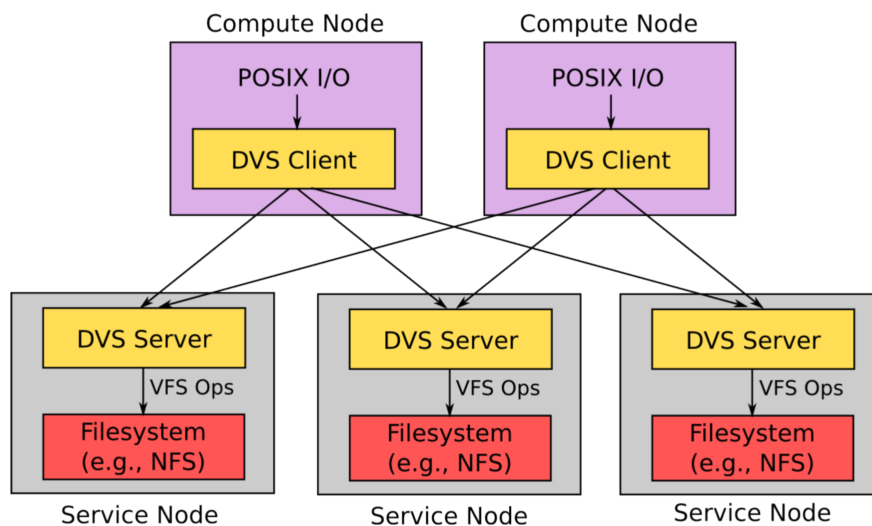

Figure 2. Overview of the DVS components. DVS clients can communicate with multiple DVS servers over the network. The lower filesystem on the DVS servers handles coherency.

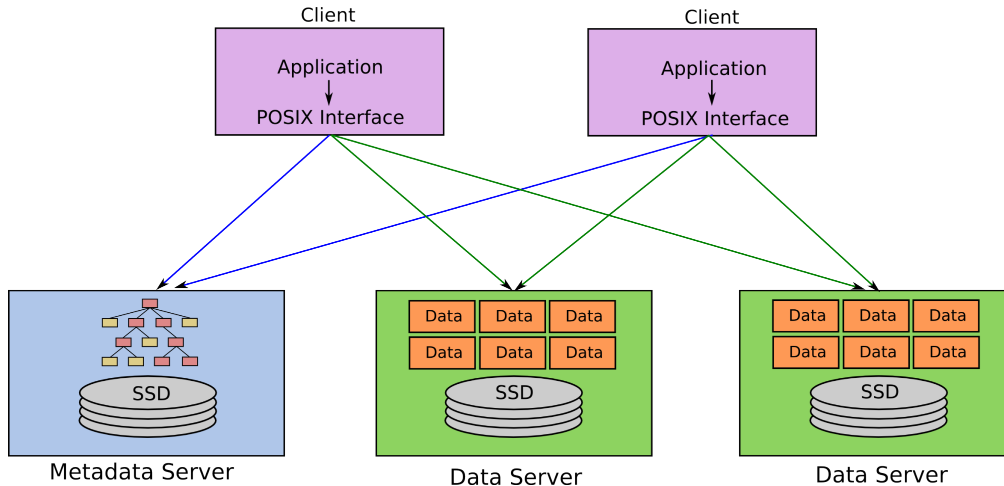

Figure 3. Layout of DataWarp clients, metadata server, and data servers. Metadata requests from the client target the metadata server, and data requests target the data servers.

III. Scratch Data Path

The DataWarp scratch data path is a distributed filesystem that allows applications to see the separate SSDs on DW servers as a single filesystem. This section covers the implementation of the distributed filesystem that the DataWarp team developed specifically for this purpose.

A. Overview

The DataWarp scratch data path can be divided into three main areas: client, metadata server, and data server. Clients are the end users of the filesystem where a POSIX I/O interface is provided for interacting with files. The metadata server is where all client based metadata operations are directed, and the data server is where all client based data operations are directed. Figure 3 shows how these components are connected.

Each DataWarp scratch filesystem only has a single metadata server. The metadata server has an on disk directory structure representing the namespace of the filesystem. Unlike a local filesystem, however, the inodes in the directory structure do not contain data. They are a skeleton structure that clients perform metadata operations on only.

The data associated with each metadata inode is striped across one or more data objects on the data servers. The striping is done using a round robin algorithm based on a configurable block size, data object count, and starting stripe. The starting stripe is chosen using the inode number of the metadata inode, and it determines which data server the first data object is placed on. There is no replication of data, so each data object contains a unique subset of the full file's data.

The clients communicate with the metadata and data servers using DVS. The DataWarp environment is significantly different that the typical parallel filesystem DVS was designed to project, so new mount options were added to alter its behavior. These mount options include the ability to send metadata and data operations to separate servers. DVS uses the metadata information from the metadata server and the data from the data servers to piece together the full view of a file on the client.

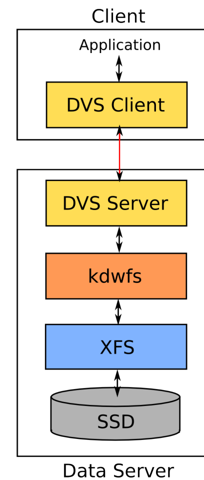

Figure 4. Filesystem stack from the client to the server for DataWarp scratch.

B. Filesystem Stack

The SSDs on the DataWarp servers are formatted with XFS. kdwfs is mounted on top of the XFS mount, and DVS projects the kdwfs mount to the DVS clients. Figure 4 provides a graphical view of the filesystem stack.

kdwfs is a stackable filesystem that provides the ability to separate a file into its metadata and data components, and it can spread them across multiple nodes. The metadata and data components are simply inodes that kdwfs stores on the lower XFS filesystem. A kdwfs mount appears at first glance

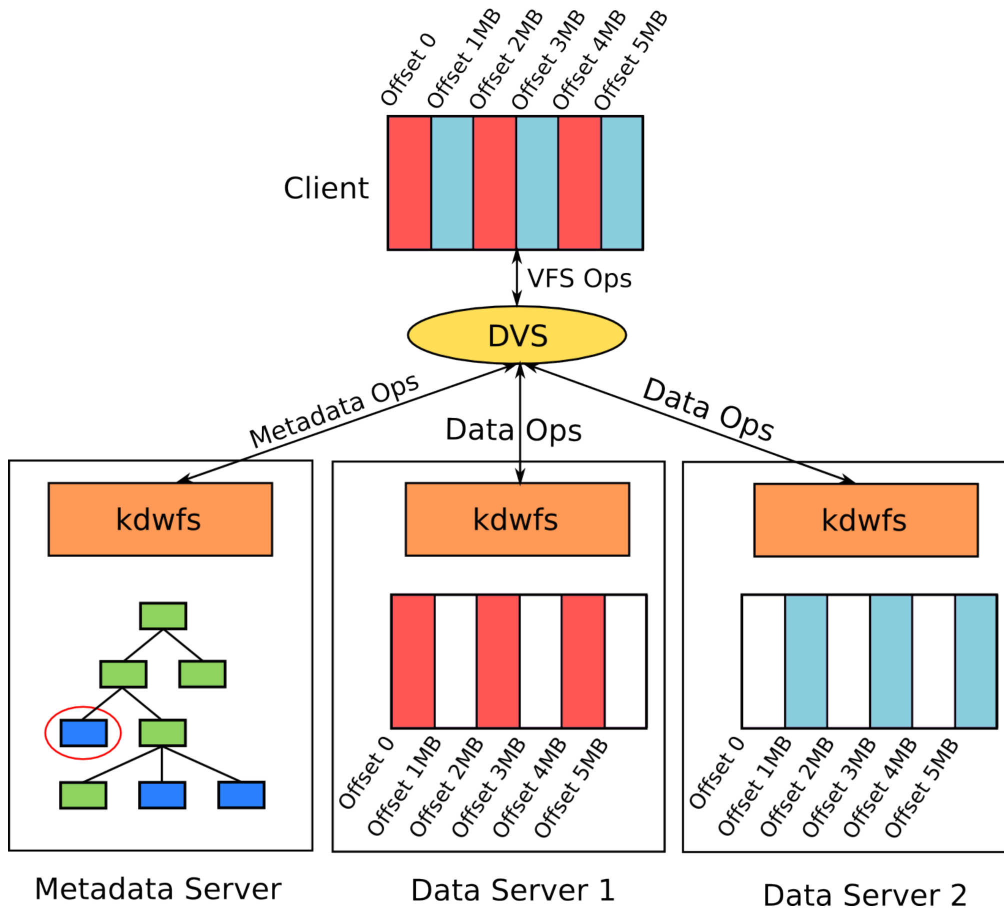

Figure 5. DVS is used to combine information from the metadata and data servers to give a full view of the file on the clients.

to have a normal filesystem layout, but its interface is not POSIX compliant. kdwfs requires DVS to stitch together the correct information from the metadata and data components from different servers.

Internally, kdwfs classifies each of its inodes as either metadata or data. The type is inherited from the parent directory that the inode is created in. The root of the kdwfs mounts on the data and metadata servers is manually set to the correct inode type at mount time. Each of these inode types only implements a subset of the VFS operations. In general, the metadata inodes implement the metadata operations (i.e., inode operations) and the data inodes implement the data operations (i.e., file operations).

The fragmented nature of the inodes that kdwfs provides is depicted in Figure 5. This figure shows a DataWarp scratch configuration with one metadata server and two data servers. Looking at the metadata server first, the kdwfs mount shows a directory tree with a single inode highlighted that we are interested in. Each of the data servers shows the contents of the data inode that is associated with it. In this case, the data is striped across the two servers on 1MB boundaries. It is clear that viewing the inodes directly through the kdwfs mount does not give an accurate view of the whole file. However, looking at the client view, DVS gives the expected information by combining parts from all three of the DataWarp servers.

Since DVS has to combine information from the metadata and data servers, it needs to be aware of the file layout that kdwfs provides. When the DVS clients are mounted on the compute nodes, one of the mount options is to specify which server is the metadata server. Each DVS client will forward all metadata operations to that server where they interact with the kdwfs metadata inodes. When an application does an `open()`, the DVS client forwards that request to the metadata server. All permissions checks are done on the metadata inode, and if the open succeeds, the DVS server queries kdwfs through an `ioctl()` about where to find the associated data objects. kdwfs provides a path name and a server for each of the data inodes, and the DVS server passes that information back to the DVS client.

The DVS client does not immediately open the data inodes. In fact, some of the data inodes may not even exist yet. Instead, DVS will defer the open of a data object until the application does a data operation (e.g., `write()`) with

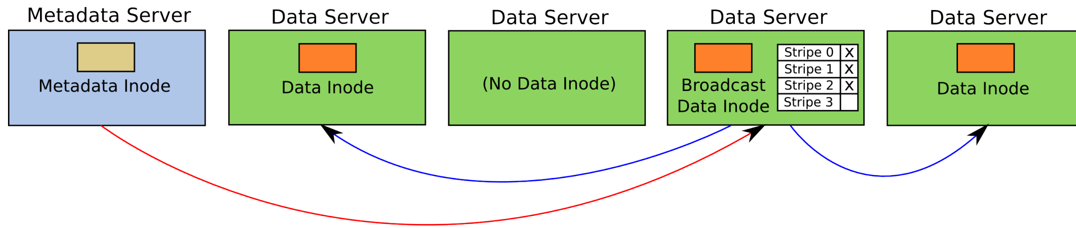

Figure 6. Network broadcasts from the metadata server require two hops. The metadata server contacts the data server with the broadcast data inode, and that data server broadcasts in parallel to the other data inodes that are present.

a range that includes that data object. At that point the DVS client will open and potentially create the kdwfs data inode. This allows DataWarp to avoid work when accessing small files whose data objects only need to cover a subset of the data servers. Once the DVS client has opened a data inode it can send data operations to that server without any further contact with the metadata server. When the client closes a file, it sends the close to the metadata server and all of the data servers.

C. Network Communication

DVS is responsible for piecing together information from each of the DataWarp servers so applications can interact with a POSIX filesystem. However, there are also situations when the DataWarp servers need to communicate between each other. This is the responsibility of kdwfs.

The data inodes in kdwfs are always associated with a metadata inode on the metadata server. The metadata inode has to manage its data inodes since many metadata operations can have an effect on the data as well. For example, a `setattr()` can truncate a file which requires truncating the data inodes too. The metadata inode has to coordinate with the data inodes by sending messages over the network to the data servers. Messages between the metadata inode and the data inodes always originate on the metadata server and flow out to the data servers.

The most expensive operations in kdwfs are those that require network communication between more than two DataWarp servers. When DataWarp is running across hundreds of servers, a metadata inode broadcasting to all of its data inodes can add a significant amount of overhead. The network requests have some parallelism to them, but the broadcast time still scales linearly with the number of servers to contact. kdwfs tries to reduce this performance bottleneck by decreasing the number of servers it has to contact on broadcasts. One way this is done is by tracking which data inodes exist for a particular metadata inode and only broadcasting to those servers. This is possible since the creation of individual data inodes is deferred until they are needed, so small files will not have data inodes spanning across all the data servers.

A central location is needed to track which data objects are present for a particular metadata inode. When a new data object is created, it will announce its presence by sending a *notify_create* message to that location, and it will be included as a candidate for subsequent broadcasts. The metadata server is the natural place to store this tracking information since most broadcasts originate there. However, the metadata server would become a bottleneck due to all the *notify_create* messages targeting it. This problem would be especially bad during a file-per-process workload. To avoid this bottleneck, each metadata inode designates one of its data inodes to store the tracking information. This inode is called the broadcast data inode. The broadcast data inode is stored on a different server from the metadata inode, and the server is picked based on a hash of the metadata inode number. This distributes the network load generated by *notify_create* messages across all the data servers during file-per-process workloads.

Most operations that require a broadcast originate from the metadata server where the tracking information is not available to efficiently target the data inodes. Therefore, network broadcasts are actually a two step process. The metadata server first sends a message to the data server that holds the broadcast data inode. That server processes the network request locally and uses its tracking information to broadcast to all the other data servers with data inodes. Figure 6 shows how the network broadcast travels between the different servers.

Some of the common operations that require network messages in kdwfs are listed below.

1) `create()` When a data inode is created, kdwfs sends a message to the broadcast data inode so the new data object can be tracked.
2) `unlink()` kdwfs will unlink all of the data inodes after a metadata inode has been unlinked.
3) `getattr()` kdwfs collects size and time attributes from the data inodes when a `getattr()` is called on the metadata inode.
4) `statfs()` The global filesystem stat needs to combine filesystem information from all of the individual disks.
5) `read()` A read past the end of a data inode triggers a broadcast to all the other data inodes to collect size information. This is needed to determine if the actual EOF was reached.

D. Attribute Caching

One of the most common sources of network broadcasts is the `getattr()` call which typically requires a full broadcast to all of the data inodes. kdwfs has a few ways of reducing the latency of this operation.

In some situations, the broadcast can be avoided entirely by caching the latest attributes on the metadata server. After a `getattr()` broadcast is done, the metadata attributes are set to match the global state of the file. Subsequent `getattr()` requests can use the local metadata inode attributes rather than broadcasting to the data inodes again. However, this is only possible until the metadata inode is opened with write privilege. At that point, a DVS client could open one of the data inodes and change its attributes through a data operation. DVS clients always open the metadata inode first, so kdwfs on the metadata server is aware of when it can use the cached attributes.

Attribute caching on the metadata server is very fast since it does not require any network communication between the DataWarp servers. However, once the inode has been opened with write permissions, the metadata server is forced to contact the data servers to get accurate size information. This is problematic for applications that open a file with `O_RDWR` or `O_WRONLY` and then immediately call `stat()`. Even though the application has not done anything to change the file size, kdwfs still has to broadcast to all data inodes. We improved this situation by adding another level of caching on the broadcast data inode. The broadcast data inode caches attributes and tracks which of the other data inodes have been opened with write permissions. When the broadcast data inode gets a `getattr()` message, it only broadcasts it to the data inodes that have been opened for writing. Whenever DVS opens a data inode with write permissions, kdwfs sends an asynchronous message to the broadcast data inode to update the tracking table. Since DVS defers the opens of data inodes until they are needed, this can give a significant performance improvement for `stat()`s that are done immediately after an `open(O_RDWR)`.

Caching file attributes will only increase performance when there are multiple `getattr()` calls to the same metadata inode while the cache attributes are valid. An initial broadcast is needed to collect the attributes from the data inodes so the result can be cached. kdwfs tries to hide the latency of this by speculatively broadcasting `getattr()` requests to the data inodes and caching the results on the broadcast data inode and metadata server. The speculative broadcast is done asynchronously to any user access, and it is triggered when the last writable file is closed on an inode.

IV. Adding Transparent Caching

The DataWarp scratch filesystem provides a simple distributed filesystem that leverages DVS's ability to scale to a large number of clients. DataWarp transparent caching builds on this infrastructure to create a filesystem with implicit caching of a PFS. We will look at how both the metadata and data portions of the scratch filesystem were modified for DataWarp transparent caching.

A. Metadata Server

In the scratch filesystem, the metadata server uses an XFS filesystem on the local storage to persist the metadata. For transparent cache, the goal is to project the PFS mount, so instead of mounting kdwfs on top of XFS, it is mounted on top of the PFS. This means that the directory tree that exists on the metadata server and gets projected to the clients is the exact contents of the PFS. It should be noted that just like in the case of the scratch filesystem, the directory tree on the metadata server is only used for metadata operations. Although the files in the directory tree on the metadata server do contain data, data accesses will still happen through the data objects.

One benefit we get from having metadata backed by a PFS is that the metadata is accessible anywhere that the PFS is mounted. This allows transparent cache to use multiple metadata servers. The underlying PFS will keep all the metadata servers in-sync through its own coherency model. A side effect of this is that the internal data structures within our stackable filesystem (i.e., kdwfs) can become out of sync with the PFS. As described in the section on wrapfs, stackable filesystems cannot tolerate out of band changes to the lower filesystem. For the most part, we are able to avoid this problem by having DVS forward all operations for a particular inode to a single metadata server. However, there are still situations where the PFS changes state under kdwfs. Those will be described in more detail under the challenges section.

The entire directory structure of the PFS is available through the metadata directory tree. However, data objects are only created after the first `open()` of a metadata inode, so almost all of the metadata inodes in the large PFS namespace will not have any data objects associated with them. This is an important note since some metadata operations have to adjust their behavior depending on whether data objects exist. For example, a `getattr()` must collect size information from either the PFS or the data objects. If the data objects are present then they have the most up to date size. If there are not any data objects, then kdwfs must call `getattr()` on the PFS inode.

B. Data Server

The data server changes required for transparent cache were extensive enough that a new filesystem was developed to hold the logic. The new filesystem, kernel data caching filesystem (kdcfs), is also based on wrapfs.

Figure 7 shows how kdcfs is layered between kdwfs and an XFS filesystem mounted on top of the SSD. kdwfs uses the same on disk layout for data inodes in both the scratch and transparent cache data paths. In the transparent cache

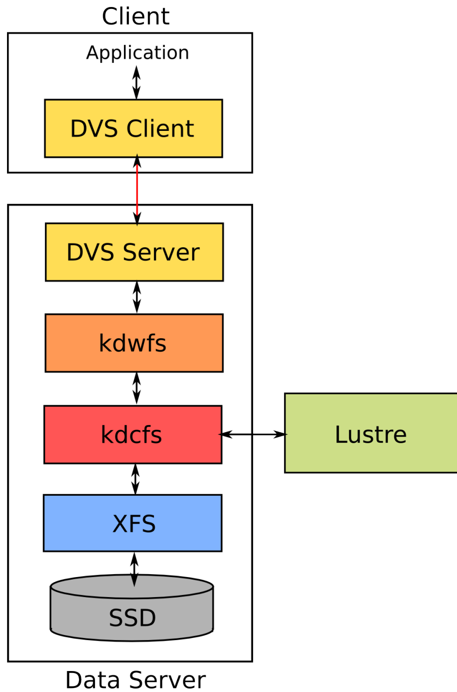

Figure 7. Data server filesystem stack from the client to the server for DataWarp transparent cache.

data path, however, kdcfs manages the data in the data inodes underneath kdwfs so that they mirror the contents of the PFS file the user opened on the metadata server. kdcfs does this by allowing each of its inodes to have an external file associated with it. The file on the SSD is referred to as the cache file, and the external file is referred to as the backing file. As Figure 8 shows, kdcfs moves data between these two files as needed. All user accesses to the data inode travel straight through to the cache file, and kdcfs ensures that the correct data is on the SSD before allowing the accesses to complete.

kdcfs provides a kernel interface for setting the backing file for an inode. kdwfs is responsible for the server-to-server communication necessary to coordinate setting the correct backing file in kdcfs on each of the data servers. This is complicated by the fact that the creation of each data inode is deferred until the overall file size is large enough to need it. Using path names to set the backing file in kdcfs opens up a timing window where the PFS file could be renamed or unlinked before all the data inodes have been created. This would lead to kdwfs setting the incorrect backing file for a cache file. To get around this problem, we use file handles to open the correct PFS file. File handles are a unique identifier of an inode within a filesystem. They can be used to open a file without having a path name, and they can be used on inodes that are unlinked but still present.

When the metadata inode is opened through kdwfs, we also query the PFS for the file handle of that inode. That

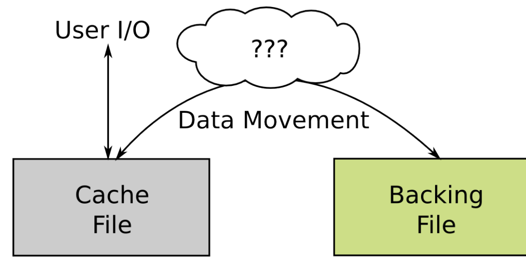

Figure 8. DataWarp transparent cache moves data between the cache file and backing file so user I/O see the same contents as the PFS.

information is passed to the broadcast data inode where it is used to set the kdcfs backing file. All data inodes that are created from that point on have to contact the broadcast data inode with a *notify_create* message. The PFS file handle is included on the reply message, and kdwfs uses it to set the kdcfs backing file on the newly created data inode. In that way, the PFS file handle is distributed from the metadata server to all of the data servers.

C. Extents

kdcfs is responsible for moving data between the cache file and backing file. All user I/O operations target the SSD, so the regions that are being accessed have to be copied from the PFS onto the SSD. kdcfs logically divides each inode into extents that have a default size of 1MB. Data is moved between the PFS and SSD on extent boundaries such that certain regions of the file can exist on the SSD. kdcfs tracks the state of an inode's extents in an in-memory radix tree which it updates based on filesystem operations. The state of each of these extents is independent, and they are tracked individually. Each extent can be in one of three states:

1) **Not-present** The data in the cache file is older than the data in the backing file

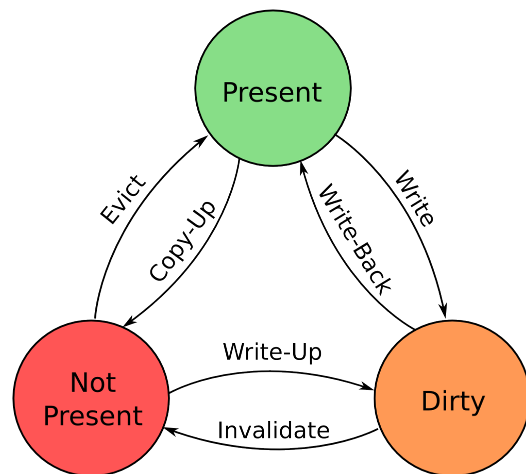

Figure 9. State diagram for kdcfs extents.

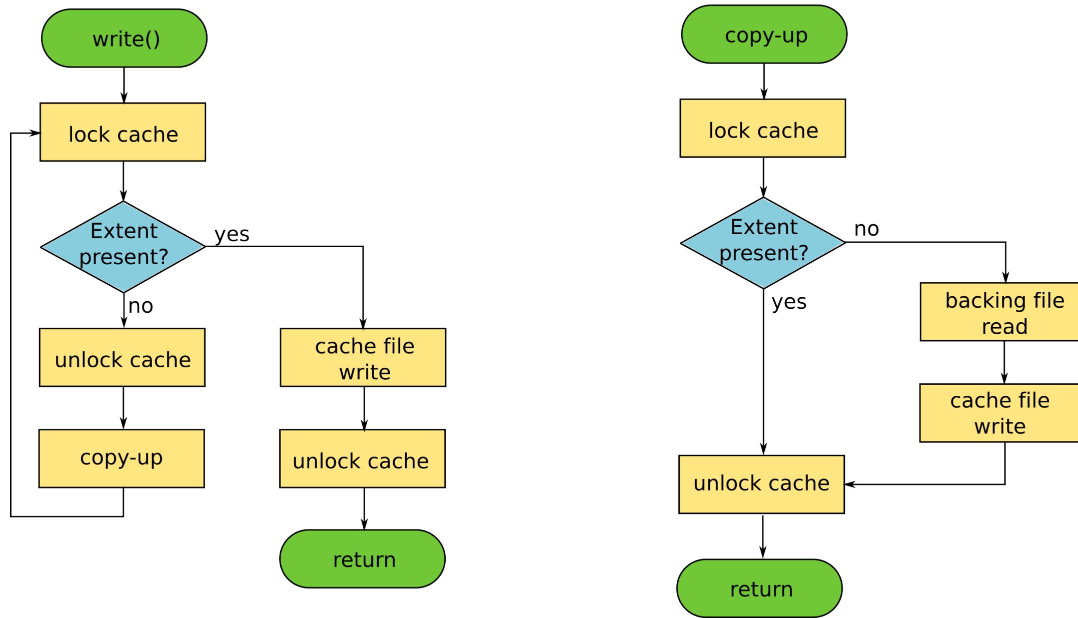

Figure 10. Left: Flowchart for user `write()` operations. Right: Flowchart for the *copy-up* cache worker.

2) **Present** The data in the cache file is the same as the data in the backing file
3) **Dirty** The data in the cache file is newer than the data in the backing file

Figure 9 shows a state diagram that models how the extent state changes based on operations to the kdcfs inode.

D. Cache Workers

Any data access to a kdcfs inode first checks the state of the extents that the data operation impacts. kdcfs will only allow access to extents that are in the *present* or *dirty* state. Data operations that target an extent in the *not-present* state must block until kdcfs copies data from the backing file into the cache file for that region. This sort of data movement in kdcfs is performed by a pool of cache worker threads. There are four operations that these workers can perform: *copy-up*, *write back*, *evict*, and *invalidate*. They are defined as follows:

1) **Copy-up** Data is copied from the backing file to the cache file
2) **Write Back** Data is copied from the cache file to the backing file
3) **Evict** The space reservation from a clean area of the cache file is deallocated
4) **Invalidate** The space reservation from a clean or dirty area of the cache file is deallocated

We can see how these workers are used by tracking a `write()` operation through the flowchart in Figure 10. We will look at a `write()` with offset 0 and size 4096. First, kdcfs uses the offset and count to find which extents are targeted. In this example, only extent 0 would be affected. kdcfs checks the state of extent 0 in the extent tree and finds that it is in the *not-present* state. A *copy-up* worker thread is scheduled, and the `write()` thread blocks until the worker thread is finished. The *copy-up* worker will copy 1MB of data at offset 0 from the backing file to the cache file. Then it will set the state of extent 0 to *present*. The *copy-up* thread finishes, and the `write()` thread is unblocked. It checks the state of the extent again and finds that the extent is now *present*. The first 4096 bytes of the extent are written to, and the extent's state is changed to *dirty*.

The example above shows how a `write()` call may trigger a `read()` from the backing file. This is necessary because the size of the `write()` only partially covered the 1MB extent. The entire 1MB extent must have a consistent state, so the *copy-up* is done to bring the entire extent into the *present* state. After the `write()` occurs, the whole extent is transitioned to the *dirty* state even though only the first 4096 bytes are actually dirty.

The *copy-up* operation will significantly impact `write()` performance since it requires interaction with the slower PFS. Performance can be improved for `write()` operations by aligning the `write()` with extent boundaries. When a `write()` covers an entire extent, it can be overwritten without copying up from the backing file first. This transitions the extent state directly from *not-present* to *dirty* without having to pass through *present*. This operation is called a write-up.

Eventually the SSD will be completely filled with data that was either written from an application or copied up from the PFS. Some of this space on the filesystem has to be freed so that the cache can be used for new incoming data. An

evict cache worker is scheduled to free individual extents within a cache file. It does this with the `fallocate()` call which can be used to deallocate blocks within a range on the lower filesystem. Not every filesystem supports this operation which is one of the reasons why XFS was picked for a lower filesystem in DataWarp. The data in the region that is deallocated is lost forever, so only extents that are in the *present* state are allowed to be evicted. The data in an extent that is *present* is already replicated by the data on the PFS, so removing it from the cache does not lose information.

Extents that are *dirty* cannot be evicted from the cache because their data is more recent than what exists on the PFS. If the underlying blocks were deallocated, the data would be lost, and the user would see corruption. *Write-back* workers are used to write the data from the cache file to the backing file for an extent. This will transition the extent from *dirty* to *present*. At that point the extent is a candidate for eviction.

E. Cache Management

Movement of data onto the SSD is driven by user events like `read()`s and `write()`s. However, determining which regions of which cache files to write back to the PFS or evict from the cache is left up to kdcfs. kdcfs has a management subcomponent that is responsible for making those decisions. The goal of the management subcomponent is to keep the correct data in the cache to minimize the number of PFS accesses that have to take place through *copy-up* and *write back*.

The management subcomponent of kdcfs chooses which extents to target for *evict* or *write back* based on algorithms within a pluggable policy layer. Eviction and write back are driven by separate policies that can be configured independently. The policies register for filesystem events (e.g., `write()` and *copy-up*) that provide information on which operations are taking place on which regions of a file. Each specific policy tracks this information in a way that makes sense for the algorithm it is implementing.

The total amount of cache space used and the total amount of data that is dirty within the cache is tracked by the management subcomponent. Each policy allows a set of high and low water marks expressed as a percentage of the total cache size that are used to determine when the policy should take action. For example, when the total cache utilization reaches the high water mark specified for the eviction policy, the management subcomponent will trigger a set of management threads that evict data from the cache until the low water mark is reached. Since eviction and write back use separate polices with different high and low water marks, they can be tuned independently.

DataWarp transparent cache currently uses a file based least recently used (LRU) policy for both write back and eviction. This algorithm works by selecting the least recently used extent within the least recently used file. The default water levels for eviction try to keep the cache utilization between 95% and 100% full, and by default write back starts when the dirty data reaches 50% of the cache size and stops when no dirty data is left.

Different workloads will benefit from different policies for both write back and eviction. In the future, DataWarp will expose a method for changing the policy to optimize for a particular access pattern.

F. Control API

The contents of the cache are modified automatically through the management policies and through user operations like `read()`s and `write()`s. In some situations, however, an external component may want to modify the contents of the cache. kdcfs provides a set of `ioctl()`s that can be used to perform *copy-up*, *write-back*, *evict*, or *invalidate* work on a file. This interface is currently used by the DataWarp service daemons, and in the future it will be exported to the compute nodes. On the compute nodes it will be available to user applications through the cache control API in libdatawarp. This will allow users to stage files to and from the SSDs in a similar way to DataWarp scratch.

V. Challenges

Adding transparent caching to DataWarp posed some significant challenges. The first challenge was that we needed a good understanding of the parallel filesystems that DataWarp would be caching. Adding a stackable filesystem on top of a parallel filesystem is not common practice, so having access to the PFS internals is important for debugging problems. For this reason, DataWarp transparent cache currently only supports Lustre as a backing filesystem.

Lustre's use of the kernel dentry cache was a source of many problems for DataWarp. kdwfs is mounted over Lustre on the metadata servers which is not an expected way to use Lustre. The dentry cache for Lustre is more complicated than for a local filesystem since the state of a dentry can change due to a remote event. For example, two Lustre clients can have copies of the same dentry, and one of those clients could rename it. The kernel VFS layer allows for this behavior with the `d_revalidate()` operation. After finding a dentry in the dentry cache, the kernel VFS layer will call `d_revalidate()` to ask the lower filesystem if the dentry is valid. However, Lustre chose not to interface with the dentry cache this way. Instead, Lustre overloads the `d_compare()` operation. Lustre prevents the kernel VFS layer from matching on the name of a stale dentry in the dentry cache. The kernel is then forced to do a real `lookup()`, but in most cases Lustre will correct the old stale dentry and return it. This behavior is non-standard, and it does not work when the wrapfs layer is inserted between

the kernel VFS layer and Lustre. We had to modify wrapfs to expect this behavior when mounted on top of Lustre.

File handles through Lustre also presented problems for DataWarp. The DW data servers use a Lustre file handle to open the backing file for their data object. Lustre file handles are not well supported, and some of the features we were expecting to get by using them did not work. We had to make several fixes to this code path in Lustre.

We also experienced some performance problems with the Linux kernel. kdcfs uses kworker threads to do I/O operations between the cache and backing files. However, the kernel throttles kworker I/O in low memory situations to work around a race condition. We were forced to use a pool of our own kthreads for doing I/O instead of being able to use the kworker infrastructure.

VI. Performance Considerations

DataWarp transparent cache has different performance characteristics than interacting with the PFS directly. Using the correct DataWarp options when creating a reservation and optimizing I/O patterns for DataWarp can help to get the fastest performance from the SSDs.

One of the biggest impacts on performance for DataWarp transparent cache is the amount of SSD space used for the cache. The cache size is specified in the user batch job script and should be matched to the I/O requirements of the application [1]. Applications that have read-only data should size the cache so that their data can fit entirely within the cache. Sizing the cache too small for a read-only workload will result in cache churn that will likely give worse performance that accessing the PFS directly. For a write heavy workload, the goal is to size the cache so that a single burst of data can fit within the cache. DataWarp will trickle the data out to the PFS, allowing the cache to be filled with new data during the next burst. Sizing the cache too small for a write-only workload will fill the cache entirely with dirty data. Incoming `write()` requests from the user will block waiting for *write back* to finish so space can be freed. This means that I/O performance will drop to the speed of the PFS.

DVS has its own performance characteristics that need to be considered since DVS is a major component in DataWarp. There is overhead when sending I/O requests from a DVS client to a DVS server, so doing small I/O limits the maximum possible bandwidth. An I/O block size of at least 1MB will give the best performance. DataWarp configures the DVS mounts with an 8MB stripe size, and a single I/O request that spans multiple servers will be sent to all servers in parallel. This means that for applications that do I/O from a single node, very large I/O sizes will give the best performance.

DVS also has a client side cache that can be configured for both read-only and read/write workloads. This improves small I/O performance, but there are several non-POSIX behaviors it introduces that make it ill suited for some applications [2].

kdwfs also introduces its own performance intricacies. As discussed already, operations that require network broadcasts between the metadata server and data servers suffer from high latency. Reducing the number of operations that trigger these broadcasts is the easiest way to improve performance at the kdwfs layer. For `getattr()`s in particular, being aware of when attributes are cached on the metadata server or in the broadcast data inode can give large performance gains. For situations where an application needs to call `getattr()` but does not require an accurate file size or modification time, kdwfs provides an extended attribute that returns a `struct stat` with invalid size and time field. This interface works similarly to a `getattr()` but avoids doing any broadcasts.

For the scratch filesystem, DataWarp also provides the ability to reduce the number of servers that an individual file is striped across. This can increase performance for file-per-process workloads since each metadata inode has to contact fewer data inodes. With a large number of user files, data will still be spread across all of the DataWarp servers, so all of the SSD space can be used. This feature is available through the libdatawarp library on the compute nodes, but it is currently only allowed in the scratch data path.

kdwfs allows data objects to map to multiple inodes on XFS. This is referred to as substriping, and it can increase the performance for single-shared-file writes. With a substripe count of 1, user threads are serialized by XFS during write operations. A single-shared-file workload will only have a single XFS inode being written to on each DW server which is not enough to drive the SSD to its full bandwidth. By substriping the data objects onto multiple XFS inodes, we gain parallelism in this situation. The substripe count for DataWarp can be configured by an administrator.

DataWarp transparent caching adds the kdcfs module which has its own preferences for how I/O should be organized. Write performance is fastest when the I/O transfer size is a multiple of the kdcfs extent size. By default this is 1MB. Writes that are smaller than 1MB will require copying data from the PFS to the SSD before the write can proceed. For `read()` operations, small I/O may see significantly worse performance than reading directly from the PFS. This is because *copy-up* will read a 1MB chunk from the PFS regardless of the user requested size.

There are also interactions at the block level that have to be considered for transparent caching. An application that writes to the cache sees the full bandwidth of the SSD up until kdcfs starts doing write back. At that point, the user `write()` threads will have to compete with the *write back* threads that are reading from the SSD. This will cause a small drop off in write performance.

Metadata performance through DataWarp transparent cache is often less than the metadata performance an appli-

cation would see directly through the PFS. This is partially because DVS is layered between the application and the PFS, and partially because there are often fewer PFS clients (i.e., DataWarp servers) than when an application interacts directly with the PFS. Applications that have high metadata requirements are not likely to be a good match for DataWarp transparent cache.

VII. Conclusion

DataWarp transparent cache provides an easy way to get improved I/O bandwidth for many existing applications. It builds off of the existing DataWarp scratch data path, so transparent cache benefits from the stabilization period scratch has had. It provides more flexibility in utilizing the SSD space since transparent cache allows access to files that are larger than could fit in the cache at one time. Also, transparent cache is easily extendable in the future to provide different cache polices to optimize the performance for individual workloads.

References

[1] B. Landsteiner and D. Paul, "DataWarp Transparent Cache: Implementation, Challenges, and Early Experience," in *Proc. Cray Users' Group Technical Conference (CUG)*, May 2018.

[2] B. Hicks, "Improving I/O Bandwidth With Cray DVS Client-side Caching," in *Proc. Cray Users' Group Technical Conference (CUG)*, May 2017. [Online]. Available: https://cug.org/proceedings/cug2017_proceedings/includes/files/pap149s2-file1.pdf