DataWarp Transparent Cache: Implementation, Challenges, and Early Experience

Benjamin Landsteiner*, David Paul†
*Cray Inc., Bloomington, MN 55425
Email: ben@cray.com
†Lawrence Berkeley National Laboratory
National Energy Research Scientific Computing Center, Berkeley, CA 94720
Email: dpaul@lbl.gov

***Abstract*—DataWarp accelerates performance by making use of fast SSDs layered between a parallel file system (PFS) and applications. Transparent caching functionality provides a new way to accelerate application performance. By configuring the SSDs as a transparent cache to the PFS, DataWarp enables improved application performance without requiring users to manually manage the copying of their data between the PFS and DataWarp. We provide an overview of the implementation to show how easy it is to get started. We then cover some of the challenges encountered during implementation. We discuss our early experience on Cray development systems and on NERSC's Gerty supercomputer. We also discuss future work opportunities.**

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

I. Introduction

Users are attracted to supercomputing platforms because they help solve their problems faster or cheaper than alternatives. As supercomputers improve, the benefits they provide increase. Advancements in supercomputing technology are not uniform, and the bottlenecks that emerge can limit a system's and user's potential. The imbalance between compute and IO performance is one area that has seen focused interest.

There are numerous proposals and deployed solutions that aim to alleviate the disparity in performance between computation and IO. Compression techniques consume some amount of excess computation and latency for increased effective bandwidth. Data caching increases bandwidth and decreases latency. On Cray systems, a recent example of a data cache is the on-node DVS client-side cache [1]. On ClusterStor, the NXD flash accelerator improves performance for small IO headed towards spinning disk [2]. DDN's IME solution has similar goals [3].

DataWarp is another solution for accelerating IO performance [4] that makes use of data caching techniques. Instead of having one shared data path for all users on a system, DataWarp can dynamically configure, prepare, and manage new data paths specific to a user's job. Since user IO requirements and properties vary, a personalized data path using the best available hardware and software can perform better. Integration with workload managers means that the sometimes critical SSD hardware resource can be shared fairly amongst a system's users. The scratch data path for DataWarp has been available for some time now with several published success stories [5], [6].

DataWarp transparent caching introduces a new data path for users to choose. Using the new data path is easier than the scratch data path because migration of data between the parallel filesystem and DataWarp SSD hardware happens transparently and automatically. Application performance is improved since data intended for the parallel filesystem is first placed on to a write-back cache backed by SSDs. We present an overview of the implementation, including the user interface, data path, and orchestration layers in Section II. In Section III we discuss some challenges encountered during the implementation. We discuss our early experience with transparent caching in Section IV. We conclude and summarize in Section VI.

II. Implementation

DataWarp Transparent Cache functionality extends existing DataWarp infrastructure [4]. With existing DataWarp, users request its functionality with `#DW` directives placed alongside workload manager directives in batch job script files. By the time the job script is executing, the allocated compute environment has been configured to access the requested SSD-backed DataWarp functionality. By tailoring the hardware and software environment optimal to the job's needs, IO performance is improved and job execution time is reduced.

The configuration steps performed depend on what the user has requested. Those steps may involve creating a storage allocation across many DataWarp nodes, instantiating a new DataWarp scratch filesystem, allocating swap files, or transferring data from the primary parallel filesystem to the DataWarp scratch filesystem. Just after compute node allocation but before batch job script execution the filesystem or swap files are made accessible to the job's compute nodes.

For the most part, cleaning up happens in reverse. Filesystem mounts and swap files are removed from the compute nodes. Important data is copied from the DataWarp scratch filesystem to the parallel filesystem. The storage allocation

is removed and the SSD space is immediately available to new users.

Users can request a DataWarp transparent cache to transparently and automatically accelerate IO to a parallel filesystem. This is in contrast to DataWarp scratch environments which requires users to manage all data exchange between it and a parallel filesystem. A workload manager (WLM) and the DataWarp Service (DWS) software work together to orchestrate the setup and teardown required for the transparent cache data path.

A. User Interface

The primary way users interact with all variants of DataWarp is through `#DW` batch job script directives placed alongside other workload manager directives for compute resources, and environment variables injected in to the batch job script execution process. For transparent cache, the `#DW jobdw` command is extended to allow users to request it. There are four mandatory parameters:

1) **type=cache** Requests a transparent caching environment
2) **access_mode=striped** Requests files be striped across the entire DataWarp allocation (i.e. over multiple servers)
3) **capacity=100TiB** Specifies the minimum size of the allocation (100TiB in this example)
4) **pfs=/pfs/path** Specifies which parallel filesystem path DataWarp will transparently accelerate (`/pfs/path` in this example)

When the batch job script executes, rather than access files under `/pfs/path`, applications can access the same files at the path defined in `$DW_JOB_STRIPED_CACHE` and the DataWarp transparent cache will work to accelerate IO operations.

The `access_mode=striped` combination can take two additional configuration options. These are specified enclosed in parenthesis and appended to the end:

1) **MFS=1GiB** Maximum File Size (1GiB in this example)
2) **client_cache** Request use of DVS client-side caching on compute nodes when using transparent caching [1]

The Maximum File Size configuration option is for helping stop errant IO activity. Since some of the hardware used by DataWarp are SSDs, and SSDs have a limited write capacity, DataWarp allows users to specify what types of IO activity may signify unintentional excessive usage. For transparent caching, a user may additionally specify two optional parameters to control how much data can be written to the SSDs in any length of time. These options have also been available to scratch environments.

1) **write_window_multiplier=5** Multiplier on capacity for amount allowed to be written (5 in this example)
2) **write_window_length=43200** Number of seconds before a write is no longer considered in calculating excessive usage (43200, or half a day, in this example)

By default, if a user does not specify these two parameters, their job is limited to 10 times the allocation size in writes per 86,400 seconds (1 day).

An example Slurm batch job script with no DataWarp integration can be seen in Figure 1. This basic job script shows an `a.out` that uses the path specified at `$JOBDIR` for IO performed during the job. In Figure 2, the script has been modified to request a 100TiB transparent cache of the parallel filesystem `/lus/global`. The path specified at `$JOBDIR` has been altered to specify use of the transparent cache. When `a.out` now runs, its IO now passes through the transparent cache rather than straight to lustre. After either job completes, the contents of `/lus/global/my_jobdir` are the same. Figure 3 shows an example with the optional settings supplied, and using `#DW` line continuation support.

While each workload manager with DataWarp support includes its own interface for showing users current DataWarp state, the DWS includes the command `dwstat` for showing current state. For transparent cache, the `dwstat configurations` endpoint includes additional entries for each transparent caching setup. See Figure 4 for an example. The primary column of interest is `backing_path`, which displays the parallel filesystem path that the transparent cache configuration will work with. Other information, such as the size and location of the allocations, are found in other unmodified `dwstat` endpoints like `dwstat instances`.

The DWS includes a separate command, `dwcli`, to allow administrators to modify DataWarp setup and manually create DataWarp environments. For transparent caching, this type of configuration can be requested. The same options that a user normally specifies with `#DW` directives in batch job scripts appear as options to `dwcli`. See Figure 5 for an example that creates a transparent cache configuration on top of the existing instance `$instance`.

All operations performed by `dwstat`, `dwcli`, and available to users via `#DW` directives are funneled through a RESTful API exposed by the DWS. For transparent caching, the `/dw/v1/configurations/` endpoint has been augmented to support its new options, such as `backing_path`.

B. Data Path

The transparent caching data path is similar to the scratch data path. The primary difference is in the introduction of a new filesystem, `dcfs` or data caching filesystem. The `dcfs` manages the contents of a buffer placed between a filesystem and a user of that filesystem. Whereas in the scratch data path the `dwfs` or DataWarp filesystem writes files to an SSD-backed XFS filesystem, in the transparent cache data

```
#!/bin/bash
#SBATCH --ntasks 3200

export JOBDIR=/lus/global/my_jobdir
srun -n 3200 a.out
```

Figure 1. Example Slurm job script with no DataWarp integration. `$JOBDIR` points to a path on lustre, so `a.out`'s IO interacts with lustre directly

```
#!/bin/sh
#SBATCH --ntasks 3200
#DW jobdw type=cache access_mode=striped pfs=/lus/global capacity=100TiB

export JOBDIR=$DW_JOB_STRIPED_CACHE/my_jobdir
srun -n 3200 a.out
```

Figure 2. Example Slurm job script requesting access to a 100TiB transparent cache of `/lus/global`. `$JOBDIR` now points to `$DW_JOB_STRIPED_CACHE/my_jobdir`. While to `a.out` the same files are opened, read, or written as in Figure 1, the IO is actually intercepted by the DataWarp transparent cache.

```
#!/bin/sh
#SBATCH --ntasks 3200
#DW jobdw type=cache access_mode=striped(MFS=10GiB,client_cache) pfs=/lus/global \
#DW       capacity=100TiB write_window_multiplier=5 write_window_length=43200

export JOBDIR=$DW_JOB_STRIPED_CACHE/my_jobdir
srun -n 3200 a.out
```

Figure 3. Slurm job script requesting transparent caching with many optional settings.

```
user@host:~> dwstat configurations --cd
conf state inst  type  amode activs backing_path
1775 CA--- 1787 cache stripe      1 /lus/snx12345
1776 CA--- 1788 cache stripe      1 /lus/global
1777 CA--- 1789 cache stripe      1 /lus/snx12345/alice
```

Figure 4. `dwstat configurations` output. The `--cd` option shows additional details on each of the transparent cache configurations. Three transparent cache configurations are viewable to the user, and each is transparently caching different paths.

```
user@host:~> dwcli create configuration --instance $instance \
               --type cache --access-mode stripe --backing-path /lus/snx12345
create request for configurations entity with id = 1776 accepted, "dwstat
    configurations" for status
```

Figure 5. Creating a transparent cache configuration with `dwcli`. Transparent cache configurations are linked with an instance, which is an allocation of space. Instances are linked with a session, which can be though of as a user.

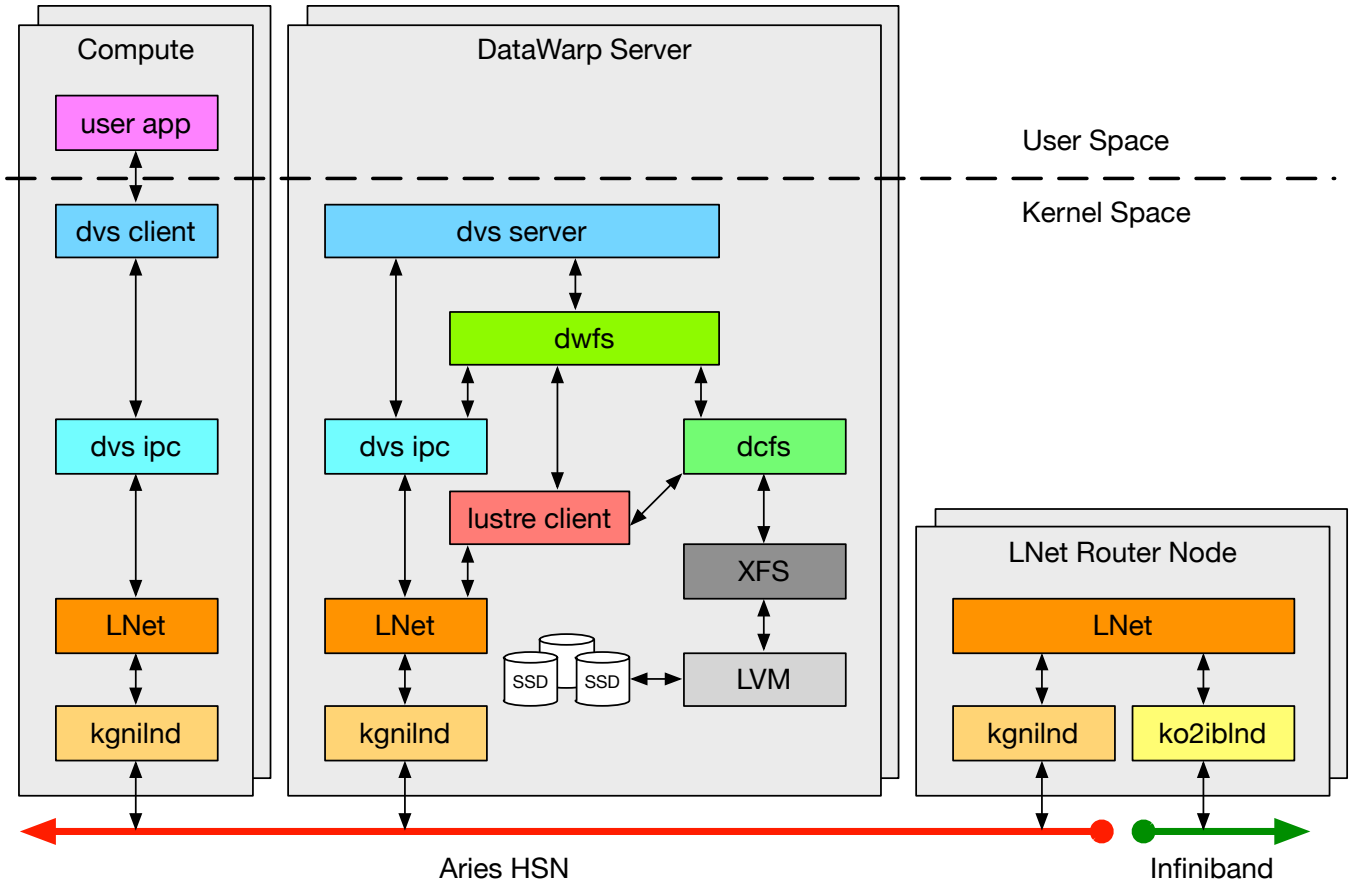

Figure 6. Visual overview of the transparent cache data path. The `dcfs` filesystem caches data on to SSD-backed XFS filesystems.

path `dwfs` writes files to the `dcfs` filesystem. The transparent caching data path is covered in depth elsewhere, so only an overview will be presented here [7]. A visual overview of the transparent cache data path can be seen in Figure 6.

On the hardware front, DataWarp server nodes are co-located with compute nodes in Cray XC cabinets. Compute nodes and DataWarp server nodes are both directly attached to the Aries-based high speed network (HSN). On the DataWarp server nodes are SSDs with an aggregate bandwidth capacity that matches the capabilities of the Aries NIC. For a filesystem external to the HSN such as the lustre-based ClusterStor, additional nodes known as LNet router nodes, which are attached to both the Aries HSN and storage network, move IO traffic between lustre clients on the HSN and the external filesystem.

The `XFS`, `dcfs`, `dwfs`, and `DVS` server reside on DataWarp server nodes. The `dcfs` automatically manages file cache data on an `XFS` filesystem placed on an LVM logical volume striped over the local SSDs. The `dcfs` manages files in 1MiB segments internally called extents. Currently, both cache eviction and dirty data write-back algorithms are file-based LRU (least recently used) and can be configured to use multiple threads. When `dcfs` processes a `read()` request, it first sees if the extents are in the cache. For any cache misses, `dcfs` performs a copy-up for these extents from the PFS in to the cache. Once all extents are in the cache, the data requested by the `read()` is returned. Processing of a `write()` is similar, as writes go through a read-modify-write cycle, except for extents wholly overwritten. Cache eviction and write-back take place automatically after a high watermark threshold of data or dirty data, respectively, is exceeded, and continue until a low watermark threshold is met. Copy-up may trigger eviction, and eviction may trigger write-back. Actual writes to the `XFS` filesystem, either through a copy-up or `write()` operation, are tabulated and used to detect potential unintentional excessive usage.

The `dwfs` manages all inter-node communication between the DataWarp servers involved in a configured transparent caching environment. The grouping of `dwfs` are called a realm. Within the realm are one or more namespaces. Each namespace is comprised of a namespace tree, where file metadata resides, and one or more namespace data repositories, where data objects reside. For the transparent caching striped access mode, there are as many namespaces

as there are DataWarp servers in the realm. Each server has a namespace tree, and each namespace has a namespace data repository on all of the DataWarp servers in the realm. This setup allows for all files accessed in the transparent caching environment to be striped across all DataWarp servers in the realm, and for each DataWarp server in the realm to service metadata operations. Since for transparent caching the namespace tree is actually a bind mount of a parallel file system mount, metadata performance is generally no better as compared to access through the parallel filesystem directly. As an example of inter-node communication `dwfs` performs, since some data may be in DataWarp and not on the PFS, some metadata in the PFS is stale. The `dwfs` intercepts user requests like `stat()` and calculates answers taking the contents of the namespace data repositories in to consideration. Another example is in forwarding user `unlink()` requests to all DataWarp servers containing stripes of a file.

The Data Virtualization Service (DVS) has a presence on both DataWarp servers and compute nodes. In short, DVS mounts on compute nodes forward IO requests from user applications over to DVS servers located on DataWarp server nodes. For DataWarp environments, DVS acts less as a generic IO forwarder and more as a DataWarp filesystem client. DVS understands the `dwfs` namespace tree and namespace data repository layout and accesses them accordingly. IO requests for the same file, or the same stripe of a file, are always forwarded to the same server. Different files or different stripes of the same file may be forwarded to other servers. This determinism comes from selecting servers based on a hash of each file's inode number.

C. Orchestration

The orchestration layers of DataWarp involve a workload manager (WLM), such as Moab/TORQUE, PBS, or Slurm, and the DataWarp Service (DWS). These components work together to offer policies regarding DataWarp usage, set up the user requested data path, migrate data between the parallel filesystem and DataWarp storage, and clean up the user requested data path.

A visual overview of the components involved in DataWarp orchestration can be seen in Figure 7. WLMs interact with DataWarp through the `dw_wlm_cli` command line client, supplying job context with each invocation. The `dw_wlm_cli` parses the supplied job script's `#DW` lines and sends requests to the DWS via a RESTful API located on a DataWarp API Gateway Node. The `dwstat` and `dwcli` command line clients are for showing status and submitting state changes, respectively, and also interact with the RESTful API. The RESTful API itself is implemented in the combination of the `nginx` web server and `dwrest` client. `dwrest` directs valid requests to `dwsd`, or DataWarp Scheduler Daemon, which persists state to a local SQLite database. Based on what a user has requested, which DataWarp servers are online, and other state, the `dwsd` will send a batch of requests to `dwmd` or DataWarp Manager Daemons located on the DataWarp server nodes. Since the batch of requests may be for performing operations on tens of thousands of nodes, the `dwmd` uses the node health fanout daemon `xtnhd` to scalably execute them in parallel [8]. The actual interactions with the data path, such as mounting, unmounting, or staging, happen via python scripts prefixed with `dws`, or just `dws*.py` collectively. User authentication across TCP/IP is performed using `munge` [9].

For transparent cache, no additional support is required from a WLM that already supports DataWarp with scratch configurations. This is because the WLM's interest involves the capacity portion of a user's request and not the way in which the capacity is to be configured. This intentional separation between DataWarp and WLM allows for new data paths to be introduced in to DataWarp with less work and coordination. Similarly, the environment variable needed by users to locate the transparent cache mount is supplied to the WLM in the same way as the scratch environment variables. The actual parsing of `#DW` is handled by changes to the `dw_wlm_cli` client.

The DWS has three main responsibilities for transparent cache. The timing of each specific step is controlled by callouts from the WLM. The first of these is the set up of the transparent cache data path. When setting up the data path, sub-steps tend to be blocked from starting on any node until prior sub-steps have completed on all nodes, e.g., mount points on compute nodes are not created until all servers have been successfully configured. The `dwsd` daemon manages the scheduling of those steps, and sends messages to `dwmd` to perform them. As with scratch, transparent cache environments are built upon DataWarp instances, which are allocations of SSD space across one or more DataWarp servers. On each allocation fragment, the DWS makes an `XFS` filesystem. This is used by the DataWarp filesystem components, `dcfs` and `dwfs`, to store file data. The DWS creates the `dcfs` mount points, via changes to one of the `dws*.py` scripts, and specifies the `XFS` mount as the device and the user-requested PFS path as a mount option. The DWS then creates a `dwfs` mount point and specifies the `dcfs` mount as the device, and the list of other servers in the allocation as a mount option. The last step on each of the DataWarp servers is to create a namespace that is configured to span each server in the allocation. The metadata directory specified at namespace creation time is a bind mount of the parallel filesystem directory which means many metadata operations actually rely on the PFS directly. The DWS sets up the namespace via a library call that sends an `ioctl()` to `dwfs`. In the final step for setup, the DWS makes a DVS mount on the nodes communicated to it by the WLM, i.e., the batch job's compute nodes. At mount time, the device is supplied as the path corresponding to the remote nodes' namespace metadata directory, and the remote nodes in the

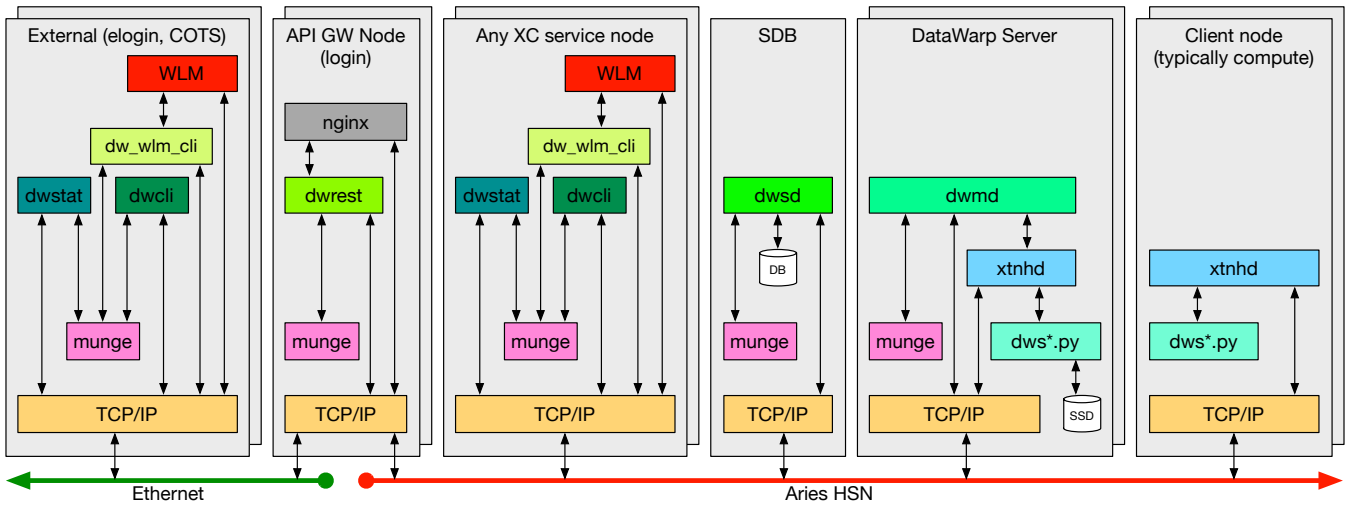

Figure 7. Visual overview of the components involved with orchestration. For transparent caching, modifications were required for `dw_wlm_cli`, `dwstat`, `dwcli`, `dwrest`, `dwsd`, `dwmd`, and `dws*.py` scripts .

instance are supplied as well.

The second responsibility the DWS has for transparent cache is managing dirty data in the cache prior to teardown. Generally, any dirty data in the transparent cache environment should be allowed to be written back to the PFS prior to teardown taking place. This is the default behavior. There are cases where the dirty data may not need to be preserved, such as a failure in the compute job or a permanent partial failure of the DataWarp instance. As with scratch, when there is interest in saving dirty data, the DWS, or `dwsd` specifically, will not allow the data path to be completely torn down until it has confirmation from it that the data has been successfully preserved. For transparent cache, prior to removing a `dwfs` namespace the DWS will call in to the filesystem by having `dwsd` dispatch a request to `dwmd`, which will invoke a `dws*.py` script, which will use an `ioctl()` targeting `dcfs` designed to both initiate write-back of any dirty data and block returning until success or error. On error, the message is logged and the operation retried after a short period of time.

The third responsibility the DWS has for transparent cache is teardown of the data path. Generally, teardown takes place in reverse order from that of setup. The DVS mount points on all compute nodes are unmounted first. Then, the previously mentioned dirty data management step takes place. It is crucial that the DVS mount points were removed first because dirty data could have been in a DVS client-side cache, or in transit between the compute nodes and DataWarp nodes. After all dirty data has been written back to the PFS or the DWS has been instructed to skip this step, server teardown steps take place. The DWS removes the namespaces on each server via a library call that sends an `ioctl()` to `dwfs`. It then removes the bind mount of the PFS used during namespace setup. The DWS then unmounts the `dwfs`, `dcfs`, and `XFS` filesystems, in that order. The DWS then performs instance removal steps as it does with the scratch filesystem.

III. Challenges

We faced a number of challenges in developing transparent caching. Our first implementation of the data path did not re-use the `dwfs` component used in the scratch filesystem. Instead, all the responsibilities of `dwfs` and `dcfs` were contained in a single filesystem layer, dwcfs or DataWarp Caching Filesystem. This led to several problems such as higher maintenance costs and higher development costs. By splitting the intra-node responsibilities (`dcfs`) from the inter-node responsibilities (`dwfs`) we were able to reduce both of these costs. Having a more modular solution also reduced the learning curve associated with working on the implementation.

Just prior to the first release of transparent caching we realized that the orchestration layer would allow users to transparently cache directories that PFS filesystem permissions would otherwise suggest the user could not access. For example, when interacting with a file path on a PFS, it is necessary to have execute permissions on all parent directories. Since the orchestration layer originally allowed a user to supply any PFS path, and the orchestration layer would then bind mount that path on to a `dwfs` namespace tree as root, the parent directory permissions were effectively ignored. To handle this problem, administrators can now configure the DWS to be more restrictive when processing transparent cache configuration requests.

Since the transparent cache data path relies on the PFS for metadata operations, it generally does not accelerate them. Since these operations must route through DVS in addition to through the PFS, the extra overhead can even result

in decreased metadata performance. We are considering improvements to DVS to reduce this overhead. Since the number of PFS clients in a transparent cache data path is the number of DataWarp servers rather than the number of compute nodes, it is possible to get better metadata performance. For large jobs where the compute node count is larger than the number of DataWarp servers, the fewer PFS clients means less coordination is required overall.

Throughout the course of development and testing we ran in to bugs that needed to be triaged and fixed. What made this trickier was running in to PFS bugs. In particular for lustre we found issues involving group lock functionality and open by handle functionality. Collaborating with lustre experts was critical in understanding and making progress on these issues.

Implementing recovery of transparent caching filesystems on server crash or reboot has also proven challenging, and is not present in the current implementation. We have found bugs in how XFS manages sparse files that prevent us from using it during recovery. While the bugs are fixed in newer Linux releases, the fixes are not easily back-ported to current CLE releases.

IV. Early Experience

A. NERSC

Implementation of Transparent Cache DataWarp (TC) for the NERSC user environment was of particular interest due to the wide range of user skill set and breadth of applications deployed on the XC40 (Cori). DataWarp usage had been well accepted by users, was stable and performant. It did however require a bit of forethought to determine input and output file requirements (names, location, size) for staging and allocation size directives. The Transparent Cache `#DW` directives simplifies the implementation for less technical users while still providing improved filesystem performance.

Cray's development of Transparent Cache was still ongoing and was not yet a released product that could be installed without risk of destabilizing the production system (Cori). Like many sites, NERSC maintains a TDS for each of the main production systems (Test and Development System). The initial installation was then targeted for the TDS. The TDS (Gerty) has a minimalistic DataWarp configuration comprising two DW-servers with 12TB of SSD. The software revisions on the TDS mirror those of the main system.

The primary purpose of the TDS is to support the installation of new OS releases or patch sets prior to roll out onto Cori. The installation of the pre-release of Transparent Cache would render this functionality ineffective due to the differences in the software levels and code differences (for example DVS). Therefore a plan was developed where both purposes could be served. A snapshot of the TDS' current SMW filesystem (BTRFS) containing the base configuration sets (cfgsets) and images (P0 images) would be created to revert back to. The TC filesets were then applied to the TDS, images generated for each type of node and another BTRFS snapshot taken. This would allow switching between snapshots, selecting which node image to boot (cnode update), and booting the TDS in either 'normal mode' or 'TC mode'. Management and coordination of this activity is aided by a 'gitflow'-like environment [10].

After rebooting the TDS with the TC-images the DWS configuration had to be restored from a JSON backup due to the change in revision levels. This involved two simple steps, one before updating and one after- using the "dwcli config" command with:

1) **backup** - backup node/pool configuration to stdout (json)
2) **restore** - attempt to restore a previously saved configuration from stdin (json)

For our initial performance testing we used two plasma physics applications; `VPIC` (writing output to a single file using the HDF5 interface) and `BD-CATS` (Big Data Clustering at Trillion Particle Scale cosmology) as shown in Figure 8. The performance returned from Transparent Cache indicates a speed up of 1.5 to 2.2 times that of the Lustre filesystem.

B. Cray

As compared to the scratch data path, the transparent cache data path includes the additional `dcfs` layer. To see the cost of this layer in the absence of write-back, we set up experiments with IOR comparing File Per Process (FPP) and Single Shared file (SSF) access patterns.

All experiments were run on an internal XC system *orion* targeting 4 DataWarp nodes. Each DataWarp node had two Intel P3608 SSDs [11]. The *orion* system was running prerelease CLE software. While the DataWarp servers were dedicated to this benchmarking effort the compute and lustre environments were shared with other users. Each DataWarp instance size was exactly 1TiB.

Each IOR run consisted of 512 compute ranks spread across 32 compute nodes. The compute node CPU types varied from run to run but were all Intel x86_64 processors, including some KNL nodes. All transfer sizes were 1MiB. For the write phase an `fsync()` was performed at the end. For the read and write tests, we used IOR's intra-test barrier option. The aggregate quantity of data written was 388GiB for the scratch vs cache comparisons and 1TiB for the write-back thread count comparison.

In Figure 9 we plot IOR read and write performance for a FPP workload using the scratch data path (dashes) and cache data path (solid). IOR reported 19,827.80 MB/sec for scratch write, and 28,921.43MB/sec for scratch read. For the transparent cache data path, IOR reported 19,381.86MB/sec for write and 28,310.38MB/sec. Comparing scratch to cache, cache was 2.2% slower for writes and 2.1% slower for reads.

In Figure 10 we plot IOR read and write performance for a SSF workload using the scratch data path (dashes) and cache

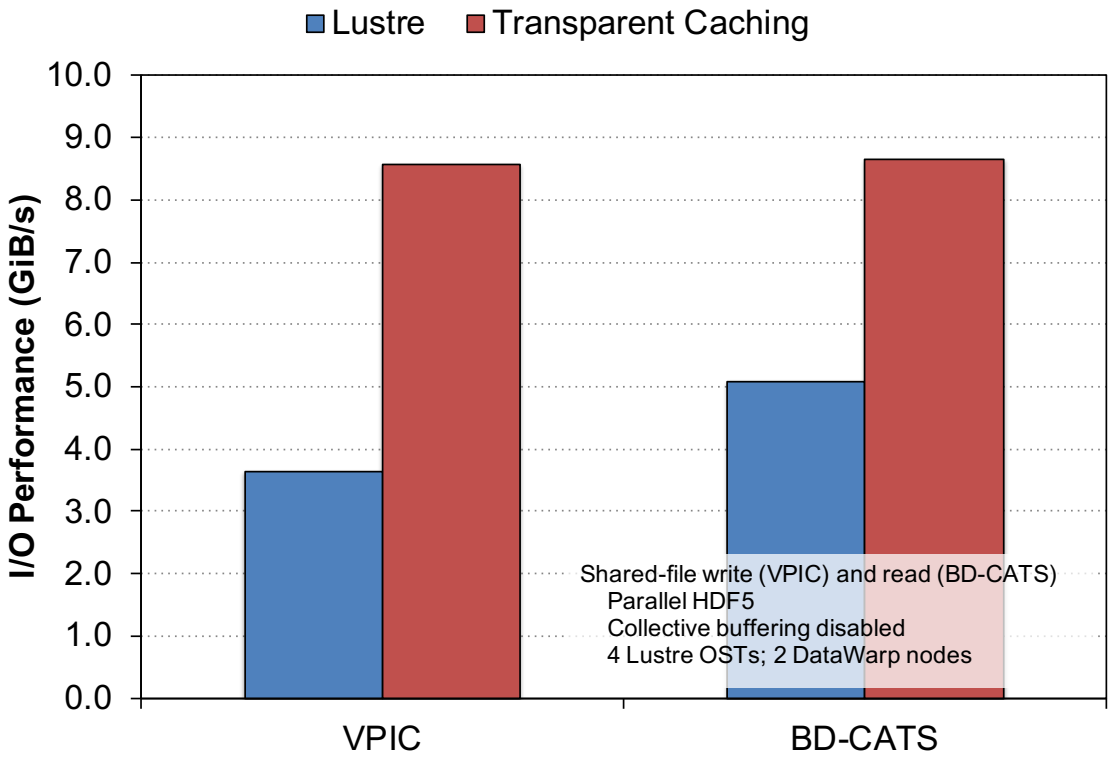

Figure 8. Initial performance results with `VPIC` and `BD-CATS`. Courtesy Glenn Lockwood, NERSC.

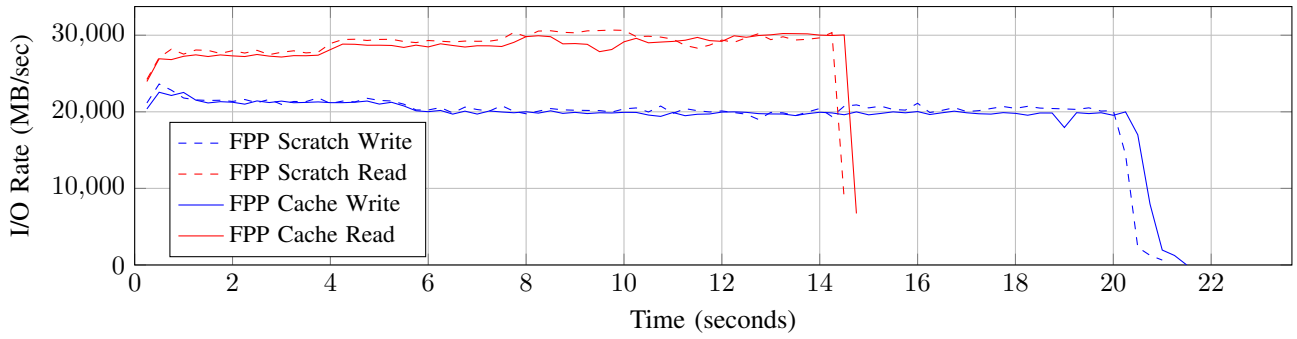

Figure 9. I/O rate comparison between scratch and cache with a File Per Process workload.

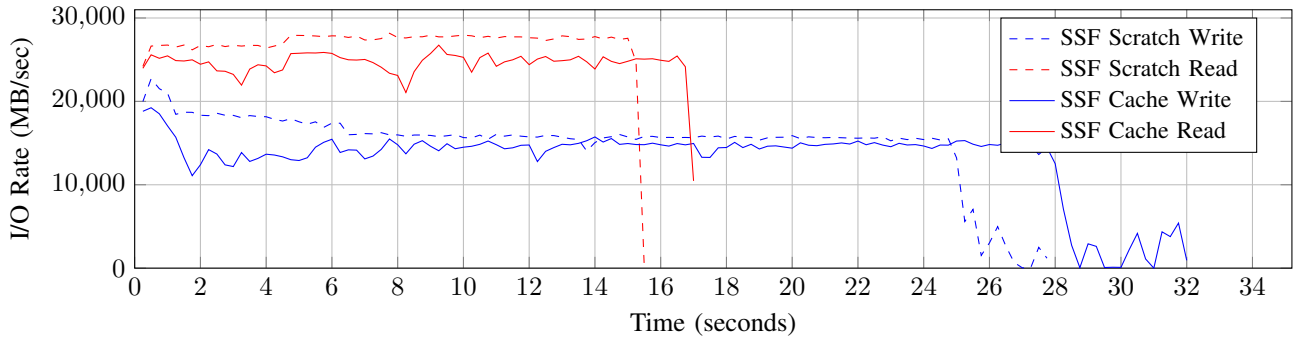

Figure 10. I/O rate comparison between scratch and cache with a Single Shared File workload.

data path (solid). IOR reported 15,055.83MB/sec for scratch write, and 27,189.43MB/sec for scratch read. For the transparent cache data path, IOR reported 13,040.54MB/sec for write and 24,503.36MB/sec for reads. Comparing scratch to cache, cache was 13.4% slower for writes and 9.9% slower for reads. The `dwfs` layer uses a sub-striping technique to improve SSF performance, but we have not yet tuned it for the transparent cache data path. We believe this explains some of the performance difference.

For transparent cache, automatic write-back begins once an internal high water mark is exceeded. For the version of DataWarp used in benchmarking, this threshold is exceeded once 45% of the data in `dcfs` is dirty. In Figure 11 we compare the impact on performance for a FPP IO pattern when there are 1 thread per server performing write-back versus 8 threads per server performing write-back. At approximately 23 seconds in both runs begin write-back of dirty data to *orion*'s parallel filesystem. When write-back occurs, these threads compute for CPU, SSD, and network resources and slow down write performance from an application's perspective.

With only 1 write-back thread, once write-back begins performance degrades from about 20GB/sec to 18GB/sec. At around 53 seconds application IO performance plummets to 2GB/sec and remains low until finishing up around 75 seconds later. The SSD cache has filled up and the application's performance is now limited to how fast the one thread on each of the four DataWarp servers can write-back dirty data to the parallel filesystem.

With 8 write-back threads, performance degrades down to around 14GB/sec. Dirty data is more quickly moved to the parallel filesystem as compared to the 1 thread variant. At around 71 seconds the application completes writing. In this example, 8 threads were able to write-back enough dirty data to the parallel filesystem while the application was writing to the SSDs to prevent the application's performance from degrading even further as was seen in the 1 write-back thread example.

V. Future Work

Load balance access mode for transparent caching is meant to greatly accelerate read-only workloads. While the old transparent caching data path implementation supported this access mode, the new implementation does not yet do so. Rather than striping files across all DataWarp servers as in striped access mode, with load balance access mode files are duplicated on each DataWarp server. Then, each compute node forwards all IO requests to just one of the DataWarp nodes. When many ranks all need to read the same file simultaneously, the full bandwidth from all servers is used immediately.

While transparent caching does automatically copy up data from the PFS in to the transparent cache on demand, it can still be useful to preload the cache before a batch job starts. This is similar to the scratch data path environment, where files can be staged (copied) on to the SSDs prior to when a batch job script executes. For batch job scripts that start out reading a large input file such as a checkpoint or database, copying up the file in to the cache beforehand is beneficial. We envision this as new batch job script directive, `#DW preload`.

Similarly, while transparent caching does automatically manage the contents of the cache, if given hints by a user it can perform better. By allowing users to explicitly request copy-up, eviction, write-back, or invalidation, the cache hit ratio can be improved and batch job execution time can be decreased.

The thresholds used by the current transparent caching eviction and write-back algorithms are preliminary and not optimal for all batch jobs. With more experience and empirical data, we will change these thresholds to be better for jobs on average, and allow for per-job configuration of these values.

The old transparent caching implementation exposed statistics or accounting data via a C library API, but the new implementation does not expose this information. The data is useful in seeing how effectively the transparent cache operated, such as seeing the cache hit ratio. The information

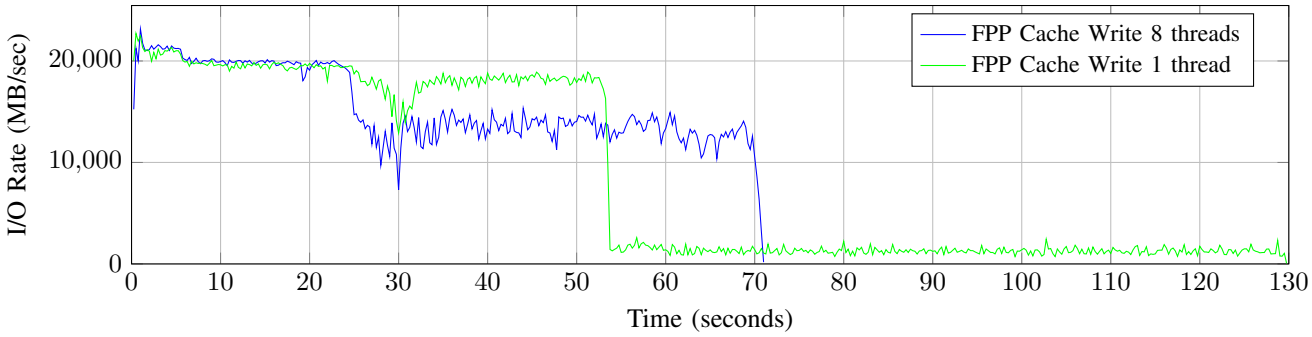

Figure 11. I/O rate comparison in the presence of write-back with 1 or 8 write-back threads.

can be used to inform increasing or decreasing the size of the transparent cache environment.

The default behavior for `close()` and `fsync()`-like operations in transparent caching is to sync all dirty data to the SSDs. Allowing this behavior to be configurable, e.g., so that all dirty data is guaranteed to have been written to the PFS on successful completion, will allow for users to choose between speed of operation and having a simple method for guaranteeing durability of data.

VI. Conclusion

DataWarp provides a scalable, highly performant I/O system for the challenges of science at scale. Technically capable scientists can take full advantage of the raw performance delivered with a few carefully placed job script directives and modifications. For those scientists that are not technically savvy the new transparent cache data path gives them access to improved I/O performance by easily fitting in to their existing scientific workflows; no explicit stage-in or stage-out of their datasets is required. As the transparent caching data path is tuned and matures, the performance gap between it and the scratch data path will narrow. Other future work will help improve read-only workloads and give sophisticated users more control over their transparent cache data path.

Acknowledgment

The authors would like to thank their teams and users at NERSC and Cray as well as those working in support of DataWarp at Adaptive, Altair, and SchedMD.

References

[1] 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

[2] Cray I/O Solutions: NXD. [Online]. Available: https://www.cray.com/sites/default/files/resources/Cray-EB-IO-Solutions-ClusterStor.pdf

[3] Infinite Memory Engine. [Online]. Available: https://www.ddn.com/download/resource_library/whitepapers/ddn_whitepapers/DDN-Whitepaper-IME-Scale-Out-Cache-Flash-Era-v3.pdf

[4] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, "Architecture and Design of Cray Datawarp," in *Proc. Cray Users' Group Technical Conference (CUG)*, May 2016. [Online]. Available: https://cug.org/proceedings/cug2016_proceedings/includes/files/pap105s2-file1.pdf

[5] W. Bhimji, D. Bard, M. Romanus, D. Paul, A. Ovsyannikov, B. Friesen, M. Bryson, J. Correa, G. K. Lockwood, V. Tsulaia, S. Byna, S. Farrell, D. Gursoy, C. Daley, V. Beckner, B. V. Straalen, D. Trebotich, C. Tull, G. Weber, N. J. Wright, K. Antypas, and Prabhat, "Accelerating Science with the NERSC Burst Buffer Early User Program," in *Proc. Cray Users' Group Technical Conference (CUG)*, May 2016. [Online]. Available: https://cug.org/proceedings/cug2016_proceedings/includes/files/pap162s2-file1.pdf

[6] A. Ovsyannikov, M. Romanus, B. V. Straalen, G. H. Weber, and D. Trebotich, "Scientific workflows at datawarp-speed: Accelerated data-intensive science using nersc's burst buffer," in *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*, Nov 2016, pp. 1–6.

[7] M. Richerson, "DataWarp Transparent Cache: Data Path Implementation," in *Proc. Cray Users' Group Technical Conference (CUG)*, May 2018.

[8] J. Sollom, "Cray's Node Health Checker: An Overview," in *Proc. Cray Users' Group Technical Conference (CUG)*, May 2011. [Online]. Available: https://cug.org/5-publications/proceedings_attendee_lists/CUG11CD/pages/1-program/final_program/Monday/04B-Sollom-Paper.pdf

[9] (2018, June) MUNGE Installation Guide. [Online]. Available: https://github.com/dun/munge/wiki/Installation-Guide

[10] D. Jacobsen, R. Kleinmen, and H. Longley, "Managing the SMW as a git Branch," in *Proc. Cray Users' Group Technical Conference (CUG)*, May 2018.

[11] Intel®SSD DC P3608 Series. Intel Corporation. [Online]. Available: http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-dc-p3608-series.html