This CUG paper is a preprint of the final paper published in the CCPE Special Online Issue of CUG 2017 at http://onlinelibrary.wiley.com/doi/10.1002/cpe.v30.1/issuetoc

Improving I/O Bandwidth With Cray DVS Client-Side Caching

Bryce Hicks
Cray Inc.
Bloomington, MN USA
bryceh@cray.com

***Abstract*—Cray's Data Virtualization Service, DVS, is an I/O forwarder providing access to native parallel filesystems and to Cray DataWarp application I/O accelerator at the largest system scales while still maximizing data throughput. This paper introduces DVS Client-Side Caching, a new option for DVS to improve I/O bandwidth, reduce network latency costs, and decrease the load on both DVS servers and backing parallel filesystems. Client-side caching allows application writes to target local in-memory cache on compute nodes. This provides low latency and high throughput for write operations. It also allows aggregation of data to be written back to the filesystem so fewer network and parallel filesystem operations are required. Caching also enables applications to reuse data previously read or written without further network overhead. This paper will discuss motivations for this work, detailed design and architecture, acceptable use cases, benchmark testing results, and possibilities for future improvement.**

***Index Terms*—Filesystems & I/O, Performance, Caching;**

I. Introduction

As the computational capabilities of large-scale HPC systems continue to improve, filesystem performance and I/O bandwidth are increasingly becoming the bottleneck to application performance. To address this imbalance between compute and I/O performance new technologies are being utilized. I/O forwarders in particular are becoming necessary to bridge the gap between the potential scale of compute resources and the scalability of available parallel filesystems. However, I/O forwarders are subject to many of the same performance concerns as distributed parallel filesystems.

Cray's Data Virtualization Service[1], DVS, is a widely used I/O forwarder on Cray systems. It is an in-kernel service, providing transparent access from compute nodes to native filesystems on service I/O nodes and to Cray DataWarp application I/O accelerator storage via Cray system's high-speed network. DVS has been proven to scale to tens of thousands of compute nodes while driving many I/O workloads at near the maximum speeds permitted by the network. There is a subset of application I/O patterns that have more difficulty reaching these speeds though. These patterns include small random reads/writes, repeating reads, and interleaved reads and writes. The difficulty with these patterns is that every filesystem operation from a compute node DVS client requires network communication with a DVS server, introducing the overhead of network latency to each operation. This is a general issue shared across I/O forwarders as well as network filesystems. In the case of small I/O transfers the latency costs associated with the network communication are disproportionate to the actual amount of data being moved. This decreases the overall I/O bandwidth.

In order to mitigate potential bandwidth issues with such I/O patterns, this paper introduces Cray DVS Client-Side Caching. Client-side caching is a new option provided by DVS to improve I/O bandwidth, reduce network latency costs, and decrease the load on both DVS servers and backing parallel filesystems and storage. It is implemented as a write-back type of cache. This option allows application writes to target local in-memory cache on compute nodes. This provides low latency and high throughput for write operations. It also allows aggregation of data to be written back to the filesystem. Aggregation of write data allows the DVS client to wait until a more optimal amount of data needs to be written and to then write all the data via a single network operation rather than a single operation for each write performed by an application. This causes fewer network and parallel filesystem operations to be required and allows for better utilization of the network fabric. Caching also enables applications to reuse data previously read or written that is available in the local client cache without incurring further network overhead costs.

DVS write caching does not provide perfect cache coherency across disparate client nodes. Instead, it provides a close-to-open consistency model (similar to NFS[2]) where dirty file data in the cache is only guaranteed to be flushed to the backing server when a file is closed, and cache revalidation occurs when a file is opened. The combination of client-side caching and the relaxed coherency between compute nodes provides the greatest possible performance benefit, particularly for the I/O patterns discussed above. However, the close-to-open coherency model will not be acceptable for all applications or I/O patterns. For example, file-per-process applications would be expected to always be safe. Applications requiring immediate visibility of data written from other client nodes may not be however. However, there are methods available that applications can take advantage of in order to avoid any potential coherency issues.

II. Motivation

DVS is capable of driving I/O at native network bandwidth and generally does a good job of maintaining this high I/O throughput over the network. However, it can be

difficult to drive applications with I/O patterns making many small random reads or writes consistently at these higher bandwidth speeds. The reason these I/O patterns are more difficult to drive at these higher speeds is that in a DVS client filesystem mount every application filesystem access requires network communication with a DVS server. This introduces the overhead of network latency to the filesystem operation. The cost of network latency can become more apparent with small-sized I/O operations where only small amounts of data are transferred in comparison to the cost of the network access, hurting overall I/O bandwidth. This overhead can be compounded by the fact that DVS had no ability to aggregate separate application I/O operations on a writable DVS client mount, forcing individual network requests for every filesystem operation performed by an application. DVS also had no means of reusing data that has been received on or originated from a client which means that multiple network operations may be required even when accessing data that has already been previously read by or written from a client. Random file accesses also prevent other performance optimizations such as file read-ahead from providing any performance benefits. With the advent of products such as Cray DataWarp[3] that leverage low-latency flash technology in the HPC storage hierarchy, the costs of this network latency are becoming the largest contributor to overall filesystem operation latency.

Previously DVS only provided the ability to cache data on read-only client mounts. Enabling that caching provided the ability for clients to aggregate read requests into more optimally sized requests and allowed the reuse of data cached on client nodes without requiring the overhead of further network communications. However, not having filesystem write capabilities meant this option was not acceptable for many client mount points on Cray systems. Enabling client caching on writable mount points was not previously done due to the potential for coherency failures across multiple compute node clients accessing a shared file and the fact that the overhead costs of managing that coherency would likely cancel out any performance gains that would be seen from using caching. Nonetheless, a means of improving the I/O bandwidth of small random I/O operations and providing better use of previously accessed filesystem data on clients was desired.

III. Design Rationale and Assumptions

The use of a write-back method of caching combined with the relaxed close-to-open coherency model allows applications the best possible performance improvement for the targeted I/O patterns. The write-back cache method provides better performance than other cache write policies. For example, a write-through type cache was prototyped but did not perform as well. While a more straightforward model and simpler to implement, a write-through cache requires every application write to still be *written through* the local cache and immediately pushed to backing server storage. This provided the ability to reuse previously accessed file data from a local client cache, but did nothing to improve on the network latency cost incurred with sending every small application I/O request to the server, which is one of the largest contributors to the overall filesystem operation latencies. These other cache write policies would actually perform worse than existing DVS today in certain cases due to the overhead cost of cache management combined with the lack of I/O bandwidth improvement.

The use of the client-side caching option does assume that close-to-open coherency will be a generally acceptable consistency model for applications performing the target I/O types. Users are required to understand the potential limitations of this model and take adequate measures to ensure that a given application is acceptable for the coherency model, take steps within the application itself to enable it to control coherency on its own when necessary, or to not use the client-side cache option with applications where it is not acceptable.

IV. Detailed Design

A. Linux VFS Address Space Interface

The new DVS client-side write-back cache feature is implemented via the Linux kernel page cache and filesystem `address_space` operations interface[4][5]. The kernel address space objects and interfaces are well documented in the kernel source documentation. DVS was required to implement and set in their default `address_space_operations` struct the following functions in order to allow data to be written to the local page cache.

1) write_begin: This is the initial function called by the kernel's generic page cache write code. It notifies the filesystem to prepare for a write of a given size at a given offset. The filesystem is expected to do any internal work necessary to insure the write can complete. It also needs to lookup the targeted page cache page and return it locked to the caller. The page can be found via a call to an exported kernel function, `grab_cache_page_write_begin()`. If the page cache page is found to be new or otherwise not marked up to date any existing filesystem data for that page will need to be read in. This requires DVS to do a `readpage` operation for the target page over the network. This is to allow the written page to then be marked up to date to allow future reads of the page and to allow correct write back of the entire page if only a portion of it is written. It is possible to optimize away this read in certain cases. For example, when the entire page is going to be written such that none of the current data in the filesystem will need to be maintained for write-back. Any portion of the page that does not currently exist on the backing filesystem may need to be zeroed to prevent previously written memory from becoming visible via an access such as page mapping.

2) write_end: Following a successful `write_begin()` call and copy of user data into the given page, the kernel makes a `write_end` call. This function is required to correctly handle any writes that were shorter than originally requested via `write_begin`, to unlock and release the page cache page, and to update inode attributes as necessary.

DVS also uses this call to implement its own tracking of currently cached pages. By tracking a queue of and number of dirty pages DVS can aggregate contiguous dirty pages until an optimal number to write-back to the server is reached. When

this optimal size has been reached that block of dirty pages can be written back to the DVS server so that they can be marked clean.

3) writepage: This operation is called by the kernel virtual memory management system in order to write out a dirty cache page to the backing filesystems real storage. Along with a pointer to the target page to be written back, this operation is also passed a `writeback_control` struct. The 'wbc' struct provides info on the type of write-back, data sync or flush, and details on how it is to be handled such as if the write-back can be allowed to fail or not. When called, this function may be allowed to write out pages other than the target page as well. Following write out this function is responsible for handling the page flags, marking the page as no longer dirty and clearing the write-back flag, as well as unlocking the cache page. DVS can also utilize the cached page queue data at this point to write back more than just the target page in order to make more efficient use of the network write request.

4) writepages: This is another operation called by the kernel virtual memory management system to write out dirty pages. This version is supplied with a range of pages to be written back. It is also supplied with a `writeback_control` struct. The 'wbc' again provides info on how the write-back is to be handled such as if the provided page write-back requests are required to complete as in the case of data syncing or providing a number of pages that should be attempted to be written if possible in the case of flushing for memory reclamation. Again DVS will attempt to use the available dirty page queue information to write back as much data as possible to the server per network request.

B. Close-to-Open Coherency Model

The DVS write-back cache provides a close-to-open coherency model. This means that file read operations are only guaranteed to see file data that was available on the server at the time the file was opened and write data cached on the client is not guaranteed to be written back to, and thus visible on, the DVS server and backing filesystem storage until file close time. This however does not imply that server data that is newer than file open time can not be read by the client or that some amount of client write data will not be written to the server prior to file close. Instead, this can be guaranteed only when the file is opened or closed. In order to maintain the close-to-open consistency, any remaining dirty data for a file will be written back to the DVS server at file close time and any cached data on a client will be invalidated at file open time. The standard Linux sync, flush, and invalidate operations that DVS already previously used are used to trigger any write-back and cache invalidation requests. The kernel VFS virtual memory interface receives these requests and routes the necessary requests for the target address space through the new write-back interfaces. This is a similar coherency model as is provided by NFS[6].

The kernel VFS interface provides caching on a kernel page granularity. This is a 4Kb sized page in the case of the Cray Linux Environment which is based off the SLES12 kernel. This implies that 4Kb is the smallest amount of data that is stored in cache, and that data is read from and written to the server in a minimum of 4Kb blocks. This page granularity affects coherency. Shared file access from different DVS clients will only be coherent at page cache size boundaries. If two DVS clients attempt to write within the boundary of the same page coherency cannot be maintained, even if the two clients are writing non-overlapping byte ranges within the page. This is because both clients would attempt to cache and write-back the entire page, unable to see the data from the other client and ultimately conflicting with each other when the clients write-back their cache pages. Cache access from separate clients will be able to maintain proper coherency if they maintain the cache page size boundaries. If the separate clients each only write and cache distinct pages then there will be no conflicts on the server when the pages are written back.

Beyond using file open and close requests to control coherency, DVS can provide more fine-grained control to user applications that require it. The generic Linux file operations sync, flush, and invalidate can be used by an application to force write back of data or to clear existing data from the local cache. Those operations used in conjunction with file locking can allow applications on different client nodes to maintain coherency. File locking on its own may not be enough to maintain coherency. Application directed sync and invalidate operations via operations such as madvise() or fadvise() would also be required to maintain the cache correctly.

C. DVS `inode` Attribute Handling and Revalidation

With the addition of cached write data on DVS clients it is now possible that inode attributes and corresponding cached file data on the client-side could be more current than the attributes and data the backing server has. This is a change from normal DVS operation today where the DVS server and backing filesystem are always expected to have the most current state for any given file. Because of this, changes to attribute handling were made to prevent inode revalidation on DVS clients from overwriting updated attributes with stale attribute info from the DVS server backing storage while a client has cached data and inode attributes. The attribute cache for write-caching mode is managed similarly to how it is today. All metadata operations, such as stat() requests, will still be pushed to the server first before also updating the clients local attributes. This will prevent the client from having to control pushing back all metadata operations done throughout the lifetime of an open file when the file is closed while still allowing the client attributes and cached data to remain current and valid on the client. When operations using attributes, such as a getattr request, are performed they will use the locally cached version of the attributes preventing the need for a network request to be sent every time attributes are accessed. The major difference to managing attributes for write-caching is that write operations will not be forwarded to the DVS server. Writes will initially only target the client page cache and any implicit file size changes caused by writes will only be seen on the client and will not be immediately

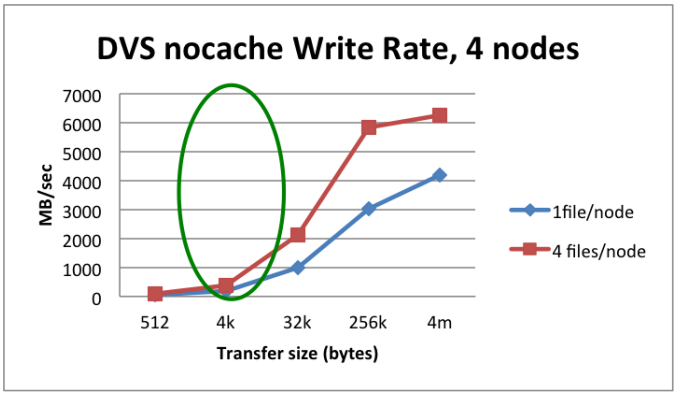

Fig. 1. bandwidth without client-side caching

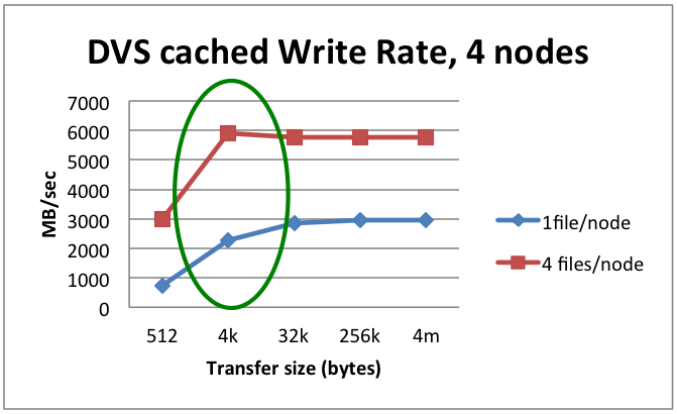

Fig. 2. bandwidth with client-side caching

seen on the server. Those pages will eventually be pushed to the server via the writepage() or writepages() write-back mechanism and will trigger any implicit file size changes on the server as the pages are written. This is a similar method to how NFS maintains close-to-open consistency.

D. Interfaces

The new write-back cache option can be controlled via the existing DVS 'cache' mount option. With the addition of the new functionality the cache option will be used to generally enable file caching, including for both reads and writes. If the existing functionality of only caching read data on a non-writable filesystem is wanted, the read-only 'ro' flag can be used at mount time to force the DVS mount to be read-only the same as it is currently when caching is enabled. The cache option will not be enabled by default. Both write caching and read only caching functionality will need to be explicitly enabled. The write-back cache option can also be controlled by applications through the existing `DVS_CACHE` environment variable as well as the IOCTL control option. This can allow an application to enable or disable the cache on a per-file or per-process basis. However, allowing an application to bypass cached data on the filesystem could lead to coherency issues between the local cached file data and what is read from the backing filesystem. This could lead to a process seeing stale file or attribute data. It is important that the application not misuse the option when caching write data.

V. Experimental Evaluation

Benchmark testing runs have shown positive results, particularly for the small-sized I/O patterns the write-caching feature was intended for. Previously DVS showed that with smaller I/O transfer sizes a lower overall I/O bandwidth speed was achieved. That bandwidth then would increase as the transfer size was increased to more optimal sizes (see Figure 1). With client-side caching enabled, DVS was able to achieve much higher bandwidth with small transfer sizes and those bandwidth speeds stayed more consistent across transfer sizes (figure 2).

I/O benchmarks intended to measure filesystem throughput have shown bandwidth increases at small I/O transfer sizes. The IOR benchmark has shown a 10x increase in small transfer size runs with client-side caching enabled. Crays internal IOPERF benchmark has shown a 100x increase in similar small transfer runs.

Customer benchmark runs have shown similar performance improvements. ToPNet is one benchmark showing a speedup. ToPNet is a single threaded application using netCDF with HDF5. It does repeated small reads and writes to a few small sized files making it a good candidate for client-side caching. One ToPNet run showed an improvement from 664 seconds without caching to only 114 seconds with write caching. A longer running customer ToPNet benchmark job running on a DVS exported GPFS filesystem improved from a 58.54 minute completion time down to a 34.84 minute completion time with client-side caching enabled. Another customer benchmark running Nastran also showed improvement. When ran on a DVS export of Crays DataWarp I/O accelerator the runtime of Nastran decreased from 9 minutes 55 seconds on a typical striped DataWarp filesystem to 6 minutes 51 seconds when using DataWarp with client-side caching enabled on the DVS mount point.

One potential downside of client-side caching that had been anticipated at design time was also proven correct by benchmark testing. It was theorized that the overhead costs of managing the page cache on clients could actually decrease the bandwidth of larger I/O transfer sizes that are already well optimized on DVS. I/O benchmarks used to test write bandwidth as transfer sizes increased showed an example of this. That test, using IOR, showed that while bandwidth increased for most transfer sizes, once the transfer size reached 512Kb the total bandwidth decreased. Further performance tuning is likely to be able to reduce or eliminate the performance difference for these larger, already well optimized transfer sizes.

VI. Opportunities for Improvements

There are some possibilities for future work to improve on the new client-side caching feature. The most advantageous improvements would be changes related to the coherency model. Changes to the coherency model in order to make

the caching feature compatible with more user applications without requiring changes or special configurations would be useful. That could prove difficult to implement however because the overhead of providing perfect coherency across large scales of compute nodes could potentially cancel out the improvements seen from caching. Other possibilities for coherency improvements could be automatically detecting when applications arent using block aligned I/O in order to disable caching automatically to maintain coherency or providing a model similar to what NFS refers to as weak cache consistency. That model enables clients to detect if they have stale file info with only minimal coherency management overhead. The existing DVS `attrcache_timeout` option could also be connected to write caching in order to make it configurable how long data should be cached and when attributes should be revalidated. That could bring the client-side caching coherency model much closer to what is provided by DVS today.

Other improvements related to performance and tuning could also be made. The most helpful would be improving performance for larger transfer sizes. Changes to the write-back heuristics in order to make the write-back transfers more similar to those done for larger write requests is likely to help performance. Another option may be to make some write-back operations asynchronous to enable the application to continue making forward progress while write-backs are performed in the background. Recent changes in the Linux kernel to allow page write-back to be performed without holding the page lock could improve performance by allowing access to cached pages even while under write-back.

VII. Conclusion

Usage of the new client side caching functionality, where appropriate, helps to mitigate some of the possible downsides of using a network I/O forwarder. It provides another tier of data storage for written file data by allowing application writes to be completed quickly to local high-speed memory. Optimized write-back of aggregated data to backing storage decreases filesystem access latency and overall network, DVS server, and backing filesystem load. Benchmark testing with client-side caching enabled has shown improvements for small transfer size writes of up to 100x as well as similar improvements for read speeds by enabling caching of read data even on writeable filesystems.

References

[1] S. Sugiyama and D. Wallace, "Cray DVS: Data Virtualization Service," in *Cray User Group Conference (CUG)*, 2008.

[2] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation or the Sun Network Filesystem," 1985.

[3] D. Henseler, B. Landsteiner, D. Petesch, C. Wright, and N. J. Wright, "Architecture and Design of Cray DataWarp," in *Cray User Group Conference (CUG)*, 2016.

[4] R. Gooch, "Overview of the linux virtual file system." https://www.kernel.org/doc/Documentation/filesystems/vfs.txt.

[5] "locking rules for vfs-related methods." https://www.kernel.org/doc/Documentation/filesystems/Locking.

[6] "Linux nfs overview, faq and howto documents." http://nfs.sourceforge.net.