Datawarp Accounting Metrics

Andrew Barry
Cray Inc.
Bloomington, MN USA
abarry@cray.com

***Abstract*—Datawarp is a Burst Buffer technology from Cray, which includes high performance flash memory storage devices, and software that allows user batch jobs to reserve some fraction of the available capacity for exclusive use. Jobs using Datawarp enjoy substantially improved filesystem performance, though with a limited capacity. However, certain user behaviors result in suboptimal performance for that user's application, and other behaviors result in degraded performance for the whole system. Thus, systems administrators benefit from Cray's tools for tracking which users are utilizing Datawarp, and how. This paper discusses how to use those tools, the data presented by the tools, and presents case studies wherein those data indicate potential usage issues by users.**

I. Datawarp Technology

Datawarp is a high performance Burst Buffer technology from Cray. The product includes mainframe service node blades installed with two or more solid state drives (SSDs), as well as software enabling compute jobs to use this storage. The SSD storage used by Datawarp is very fast, allowing only a pair of drives to saturate the network interface of the service node. This performance allows a small number of storage devices to provide a very high level of file system performance to the system. While Datawarp storage is very fast, the capacity is limited. Flash storage cannot yet completely replace spinning disk filesystems on most HPC systems. Datawarp offers a solution to this by using the SSD storage only for application data during the run-time of the compute job, not for long-term storage. Depending on the amount of available capacity, Datawarp storage may only be used by a subset of jobs that derive substantial benefit from the additional performance, compared to spinning disk storage.

Presented with the potential performance gains, many system administrators want to know how it is being used, how effective it is, and who is using the resource. Datawarp capacity is a limited resource, and systems may not have enough capacity to completely replace parallel filesystems for all storage needs. Even for systems that do contain very large amounts of Datawarp storage devices, some applications may chose to use direct parallel filesystem access instead of Datawarp. The most obvious reason for this is the wear limits attached to the flash storage within the solid-state drives. Every flash cell within a drive can only be written a limited number of times. Since Datawarp resources are finite, it is desirable to determine which applications are making effective use of the capacity. Applications that misuse the storage, or get little benefit from using Datawarp may want to continue using parallel filesystems. To answer all of these questions, Datawarp includes accounting functionality.

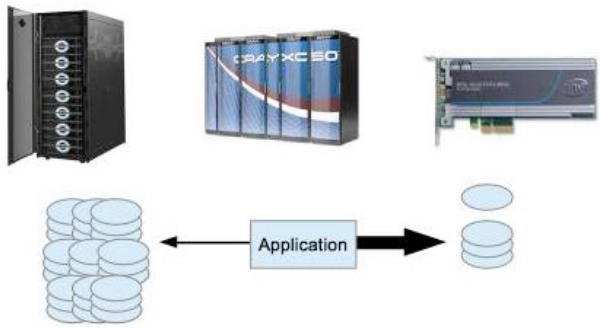

Figure 1. Parallel filesystem versus Datawarp

To understand how Datawarp accounting works, it is first necessary to understand a little more about how Datawarp works. When a user wants to use Datawarp to speed up application input/output performance, he/she must put #DW directives into the job script submitted to the batch workload manager. The workload manager will then schedule jobs to run when Datawarp capacity is available to meet the needs of the job, as indicated in the job script. When a Datawarp job is run, the workload manager will contact the Datawarp service, which then creates a filesystem for the compute job to use. This filesystem may be either a cache layer between the application compute nodes and a parallel filesystem like Lustre, or it acts as a stand-alone scratch filesystem. In either case, the filesystem is created on storage made available by some or all of the service nodes with solid-state drives, which is served to the compute nodes using Cray's Data Virtualization Service (DVS) technology.

II. Datawarp Architecture

Datawarp uses the following terminology to describe the data abstractions used in its implementation. Each time a job asks for a Datawarp filesystem to be created, a Datawarp instance is created, which is a master descriptor for all parts that constitute the filesystem. The compute job is represented by a session object, which can link to one or more instances. Each session is identified by a token, which is a text string

provided by the user or workload manager. Datawarp instances are created with a static capacity, which is

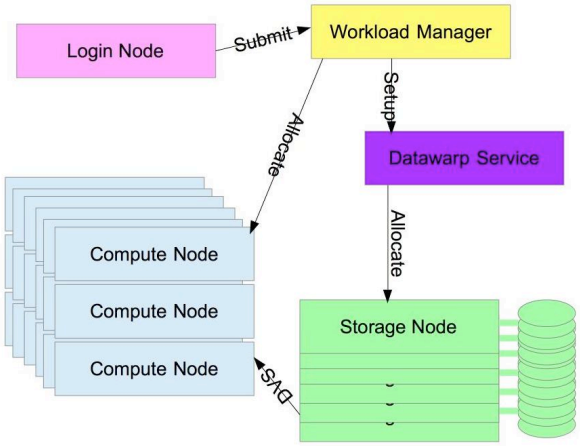

Figure 2 Datawarp Topology

composed of one or more fragments. A fragment represents the storage provided for the instance, from a single storage node. Fragments are allocated from one or more storage nodes, typically with each filesystem containing fragments from most or all of the storage nodes in the system. Scratch type filesystems may also be created with different namespace options. Compute nodes may all share a common namespace, or each get private namespaces. Thus there are one or more namespaces associated with each Datawarp instance, which Datawarp sometime describes as configurations. Each namespace is served by one of the storage nodes providing the capacity for the instance.

III. DATAWARP ACCOUNTING

The architecture of Datawarp allows for a large number of storage nodes to serve data to each application's Datawarp volume; ideally all of the storage nodes would serve data to all applications. This allows for maximum bandwidth, but presents a very complex picture for collecting utilization statistics. Each Datawarp job collects statistics from each of the compute nodes using the filesystem, and each of the storage nodes serving data fragments and namespace services for the filesystem. To make things even more complex, a job can make use of more than one Datawarp filesystem, and statistics can be gathered either for the whole job, or for each constituent application, or both. The total accounting data available for measuring Datawarp workflows is large, and may be difficult to interpret without automation.

In order to collect this large set of Datawarp statistics, Cray provides a trio of plugins for Resource Utilization Reporting (RUR). RUR is a Cray tool for collecting arbitrary utilization statistics for jobs and applications run on Cray systems. RUR runs a configurable list of data collection and post-processing plugins, to collect data on processor usage, memory consumption, energy used, utilization of accelerators, and now Datawarp usage. This data is collected from the compute nodes, and sometimes service nodes, used by the job or application before the start of execution, and again after completion. These data points are compared, and the post-processed data is output to one or more data targets, such as a text file in the user's home directory, or a log file on the system management workstation.

A. DWS RUR Plugin

In Cray's 6.0up02 release of the CLE operating system a new "DWS" RUR plugin became available. This plugin provides six simple statistics from the compute nodes in an application, providing a concise overview of the aggregate activity on the filesystem:

- Number of inodes created
- Number of files created
- Number of bytes read
- Number of bytes written
- Maximum file offset read
- Maximum file offset written

B. DWS_SERVER RUR Plugin

In Cray's 6.0up04 release of the CLE operating system two new RUR plugins became available: dws_server and dws_job_server. Both of these plugins provide the same statistics, but do so at different times. Dws_server is run at the beginning and end of an application, while the filesystem is still mounted. Collecting accounting statistics at the end of applications may miss some transactions that happen outside of user's applications. Cache type Datawarp filesystems may retain cached writes in the Datawarp filesystem, which may not yet have been synced to the parallel filesystem it is caching. Scratch type Datawarp filesystems allow users to set up lists of files which will be staged onto the filesystem before application execution, and another list of files will be staged off of the filesystem, to a permanent storage location, after the application has run. Application scope accounting will miss all of the transactions associated with cache syncing and data stage-out. The Dws_job_server plugin is run at the beginning and end of the job, after the filesystems are unmounted, and all data is synced to the backing parallel filesystem for cache-type Datawarp and all data staging is completed for scratch type Datawarp.

These two plugins collect data from all of the storage nodes serving namespaces and fragments for the Datawarp instances used by the job. The dwfs and dwcfs filesystems used by Datawarp make this easy, by providing simple interfaces to collect this data from the node, and even aggregating data from multiple nodes automatically.

C. Storage nodes report the following for each namespace served:

- Maximum file offset read

- Maximum file offset written
- Total bytes read
- Total bytes written
- Number of files created
- Total bytes staged-in
- Total bytes staged-out

D. Storage nodes also report the following for each fragment of cache type served:

- Filesystem Capacity
- Capacity high-water mark
- Window-write-seconds
- Window-write-bytes
- Maximum offset read
- Maximum offset written
- Maximum offset threshold

E. Storage nodes report the following for each fragment of scratch type served:

- Filesystem Capacity
- Capacity used
- Capacity high-water mark
- Maximum window write
- Write high-water mark
- Write moving average

IV. RUR OUTPUT

For a small Datawarp filesystem served by two storage nodes, with a single namespace, the resulting RUR output may end up looking something along the lines of this:

Uid: 16443, apid: 1050, jobid: 2546, cmdname: disk_tester, plugin: dws {“token”: “2546.sdb”, “inodes_created”: 4114, “files_created”: 4096, “bytes_read”: 281474976710656, “bytes_written”: 70368744177664, “max_offset_read”: 68719476736, “max_offset_written”: 68719476736 }

Uid: 16443, apid: 123050, jobid: 2546, cmdname: disk_tester, plugin: dws_server {[“dwtype”: scratch, “realm_id”: 657, “server_count”: 2, “namespace_count”: 1, “token”: “2546.sdb”,
“fragments”: {
“3141”: {“fragment_id”: 3141, “server_name”: “nid00343”, "fs_capacity": 8796093022208, ”capacity_used": 3518437208883, "capacity_max": 4398046511104, "max_window_write": 1073741824, "write_high_water": 4294967296, "write_moving_avg": 536870912},
“3142”: {“fragment_id”: 3142, “server_name”: “nid00344”, "fs_capacity": 8796093022208, ”capacity_used": 3518437208883, "capacity_max": 4398046511104, "max_window_write": 1073741824, "write_high_water": 4294967296, "write_moving_avg": 536870912}},
“namespaces”: {“2546.sdb_0”: {“bytes_read”: 281474976710656, “bytes_written”: 70368744177664, “files_created”: 4096, “stage_bytes_read”: 0, “stage_bytes_written”: 0, “max_offset_read”: 68719476736, “max_offset_written”: 68719476736 }}]}

The above is a very balanced configuration in which two servers are sharing the load of the filesystem equally. The report also provides both the user id and the command name. In this case, disk_tester is a shell script which writes sequential large blocks to large files.

V. ENABLING RUR COLLECTION OF DATAWARP STATISTICS

In order to collect these statistics, RUR must be configured to run the Datawarp plugins. The first step on this is to enable RUR within the Cray Alps configuration file, which is done with the Imps configuration tool cfgset update. Within the Alps configuration file, the apsys prolog entry should be set to /opt/cray/rur/default/bin/rur_prologue.py, and the apsys epilog entry should be set to /opt/cray/rur/default/bin/rur_epilogue.py. Then the RUR configuration file must be updated to include the dws plugin, the dws_server plugin, or both. The Datawarp plugins are not enabled by default; running cfgset update will allow setting each of these plugins to true.

Enabling the job scope RUR plugin, dws_job_server, is more complicated. The workload manager prologue and epilogue scripts will have to be edited to include calls to RUR, and a job scope RUR configuration file will need to be created. Firstly, /etc/opt/cray/rur/rur.conf should be copied to the workload manager configuration directory. Dws and dws_server plugins should be set to ‘false’, and the dws_job_server plugin should be set to ‘true’. Once the config file is set up, the prologue and epilogue scripts should be modified to include calls to RUR. The prologue should include:

/opt/cray/rur/default/bin/rur_prologue.py –a 0 –j $JOBID –c $CONFIGFILE –n $NIDLISTFILE –A jobfile=$JOBFILE –A jobtoken=$JOBTOKEN.

The epilogue should include:

/opt/cray/rur/default/bin/rur_epilogue.py –a 0 –j $JOBID –c $CONFIGFILE –n $NIDLISTFILE –A jobfile=$JOBFILE –A jobtoken=$JOBTOKEN.

The nidlistfile should be a file containing a list of the storage nodes, while the configfile should point to the cloned configuration file in the workload manager configuration directory.

VI. DATAWARP ACCOUNTING USE CASES

The Following are a sample of scenarios in which RUR statistics identifies a users making poor use of the Datawarp resources, or the system providing suboptimal performance. Variants of all of these have been found on test systems during Datawarp development. They may or may not be representative of the behaviors of users and software on

Datawarp systems used in production. Note that the output of the RUR plugin is in bytes. For the purpose of this paper, those values have been translated to read in gigabytes and terabytes where it improves readability

A. *Tracking Disk Writes*

The first use case for accounting is simply tracking which users are consuming the write cycles on the flash drives in the storage nodes. Workload manager reports only indicate how much space users are requesting, not how much of the drive's limited lifetime each user is consuming. Below we see that the user wrote 184 terabytes of data, which is forty-six full-drive writes spread across thirty-two drives. Simply adding up these values across all of a user's jobs shows how much of the drive write endurance that user is consuming. Systems that bill users based on compute node hours used might consider also billing based on flash drive write endurance consumed.

Uid: 16443, apid: 24104, jobid: 18052, cmdname: TDI.py, plugin: dws_server {["dwtype": scratch, "realm_id": 803, "server_count": 16, "namespace_count": 1, "token": "18052.sdb",
"fragments": {…},
"namespaces": {"18052.sdb_0": {"bytes_read": 22.6TB, "bytes_written": 184TB, "files_created": 26543… }}]}

B. *Overallocating Storage*

The next case is that of the user who was playing it a little too safe on capacity. One can see from the following RUR report that the user requested a thirty-two terabyte partition, but only used four gigabytes. Obviously there are times when this is necessary, but if a user is constantly over-provisioning the storage for their jobs, it impacts the ability of other users to run.

Uid: 16771, apid: 14654, jobid: 3116, cmdname: rzd.py, plugin: dws_server {["dwtype": scratch,
"fragments": {
"511": {"fs_capacity": 32TB, "capacity_used": 1.03GB, "capacity_max": 8TB, ...},
"512": {"fs_capacity": 32TB, "capacity_used": 1.02GB, "capacity_max": 8TB, ...},
"513": {"fs_capacity": 32TB, "capacity_used": 1.02GB, "capacity_max": 8TB, ...},
"514": {"fs_capacity": 32TB, "capacity_used": 1.02GB, "capacity_max": 8TB, ...}
}...]}

C. *Excess Staging*

The next case is that of a user who preloaded hundreds of gigabytes of data, then read up a very small proportion of that data, using the scratch filesystem primarily for writes. There are times when this is appropriate, as in importing a complex data-set, making changes in-place, and writing the final data out to permanent storage. However, in this case the data is not staged out automatically. If it is not written out as part of the user application, one must wonder if the user is wasting the resource.

Uid: 16443, apid: 24186, jobid: 18132, cmdname: postset, plugin: dws_server {["dwtype": scratch, "realm_id": 844, "server_count": 2, "namespace_count": 1, "token": "18132.sdb",
"fragments": {...},
"namespaces": {"18132.sdb_0": {"bytes_read": 2.3GB, "bytes_written": 60.73TB, "files_created": 273, "stage_bytes_read": 415.2GB, "stage_bytes_written": 0, ... }}]}

D. *Dissimilar Stripe Allocation*

The following performance issue was discovered before Datawarp accounting tools were available, but is indicative of the sort of problems many users are interested in finding with these tools. It accounts for a Datawarp filesystem that provided performance significantly less than what the user expected, given the number of storage nodes in the system. Here we notice that one node contains five times as much capacity as all of the other storage nodes serving the data. This occurred because the Datawarp service had nearly exhausted all of the available Datawarp capacity before the allocation for this instance. When this instance was created, it exhausted the capacity on seven of the storage nodes, and allocated all of the rest of the instance onto nid00221, which had more unallocated space available. Datawarp data is striped across all of the storage nodes in the instance. In this case, the striping was dissimilar between the storage nodes, and nid00221 had five times as many stripes as any of the other storage nodes. Thus the performance of the instance was limited by nid0221, which provided one fifth of the performance one would expect for each stripe, and the sum of the eight servers were able to provide only about two servers worth of bandwidth.

Discovering this problem led to the implementation of new allocation modes in the Datawarp service called 'Equalize Fragments' which over-allocates instances to prevent fragmentation. Another option, 'Equalize Fragments Guarantee' will reject allocations that would provide suboptimal performance; in such a case, the workload

Figure 3 Standard Allocations

Figure 4 Equalize Fragments Allocations

manager would re-queue the job request, and run it when capacity is available on all server nodes. Though this sounds wasteful of capacity, it is not dissimilar from filesystem settings that ensure that file allocations begin at the start of a disk stripe boundary.

Uid: 16771, apid: 12350, jobid: 3522, cmdname: HIO_wrapper, plugin: dws_server {[“dwtype”: scratch, “realm_id”: 955, “server_count”: 8, “namespace_count”: 1, “token”: “ 3522.sdb”, “fragments”: { “nid00343”: {"fs_capacity": 3TB, ”capacity_used": 155GB, "capacity_max": 250GB}, “nid00344”: {"fs_capacity": 3TB, ”capacity_used": 155GB, "capacity_max": 250GB}, “nid00345”: {"fs_capacity": 3TB, ”capacity_used": 155GB, "capacity_max": 250GB}, “nid00346”: {"fs_capacity": 3TB, ”capacity_used": 155GB, "capacity_max": 250GB}, “nid00221”: {"fs_capacity": 3TB, ”capacity_used": 750GB, "capacity_max": 1.25TB}, “nid00222”: {"fs_capacity": 3TB, ”capacity_used": 155GB, "capacity_max": 250GB}, “nid00223”: {"fs_capacity": 3TB, ”capacity_used": 155GB, "capacity_max": 250GB}, “nid00224”: {"fs_capacity": 3TB, ”capacity_used": 155GB, "capacity_max": 250GB},

VII. SUMMARY

Recent updates to Cray’s accounting tools allow the collection of statistics about each Datawarp job, and the resources it uses. This allows for studying trends in how the user base use the system, and how individual users differ from the norm. Above are a selection of user behaviors that can be identified using the collected statistics. It does not describe all interesting data patterns, and the Cray development team looks forward to seeing what other circumstance can be identified with these statistics.