Slash – The Scalable Lightweight Archival Storage Hierarchy

Paul Nowoczynski, Nathan Stone, Jason Sommerfield, Bryon Gill, J. Ray Scott
Pittsburgh Supercomputing Center
{pauln,nstone,jasons,bgill,scott}@psc.edu

Abstract

Growing compute capacity coupled with advances in parallel filesystem performance and stability mean that HPC users will inevitably create and store larger datasets. If data residing on parallel filesystems is not efficiently offloaded to archival storage, disruptions in the compute cycle will occur. Hierarchical storage caches are a vital aspect of the HPC storage machine because they offload and prime the compute engine's parallel filesystem. To keep pace with expanding HPC data systems, these caches must adapt via new scalable architectures.

The Pittsburgh Supercomputing Center (PSC) has developed a cooperative caching architecture to act as a distributed front-end cache to a hierarchical storage manager. SLASH, the Scalable Lightweight Archival Storage Hierarchy, provides means for creating parallel caching systems on an otherwise "monolithic" archival storage system without requiring modifications the archival system software.

1. Introduction

In the current realm of high-performance computing, teraflop compute systems and gigabyte per second parallel filesystems are becoming commonplace. As I/O rates to parallel filesystems increase, users will inevitably generate and store greater amounts of data. Efficient data movement to and from tertiary storage systems is crucial to proper management of parallel filesystems, which direct-attach to specialized compute resources. To support the growth of scientific datasets and capabilities of parallel filesystems, archival storage must also be given avenues for parallelism.

Traditional hierarchical storage managers consist of a single symmetric multi-processor (SMP) attached to some disks and an array of tape drives. The current scalability barrier for large SMP machines (8 processors or more) is that the cost per byte of I/O bandwidth is much higher than that of smaller, distributed, commodity-based architectures. While an SMP of this category (>8 processors) is probably needed to run an HSM managing several million files, the cost of scaling these systems is very high while the ceiling for expansion is relatively low.

Another disadvantage of most archival systems is that they do not support the variety of interconnects that are supported under linux, preventing the inclusion of the archival system into the supercomputing network. The motivation at PSC was to connect the "archiver" directly to our Quadrics-based computational interconnect, taking advantage of the high-bandwidth, remote DMA network to provide an aggregate bandwidth that scales with the number of machines on the network [2].

These factors demand a more modular expansion route that makes use of commodity hardware and linux drivers. PSC has developed a distributed caching system called the "Scalable Lightweight Archival Storage Hierarchy" - SLASH. SLASH enables the incorporation of commodity hardware components into the standard archiving scheme in a hardware layer external to the traditional archiver SMP. SLASH accomplishes this by logically extending the archiver's internal namespace to external caching nodes and performing out-of-band data migration from the caches to the archiver SMP in a coherent manner.

2. Design Goals

The primary goal of this project was to develop a distributing caching system with flexible characteristics without introducing fragile failure scenarios, static interdependencies, or large complex management tasks into the archiving system. The flexible characteristics are defined as a set of capabilities that promote scalability and ease of management.

2.1.1. Versatile Caching Facility

The distributed cache should have the ability to be grown, shrunk, or disabled easily by the administrator. This means that necessary system reconfigurations

Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2005)
0-7695-2318-8/05 $20.00 © 2005 IEEE

should not require system downtime or rearrangement of cached data or system metadata. Caching elements that leave the system should exit gracefully without causing outages to the entire system.

2.1.2. Incorporation of Commodity Hardware

In contrast to building a large fiber channel SAN, SLASH should make use of commodity storage, commodity controllers, and relatively low-cost networks like Gigabit Ethernet to build a scalable distributed storage cluster.

Steering archival design toward networked, disk-based systems [2] heeds popular arguments that tape-based archiving is already obsolete with the advent of high-volume, low-cost IDE-RAID disk solutions. Balancing this trend is the fact that the comparative cost of raw storage is still in the favor of tapes [1, 6]. By using SLASH for distributed disk storage management and archive integration, we are well situated to take advantage of either storage media as these future trends become present reality.

2.1.3. Caching System Operations are Transparent

Applications running on the caching nodes should expect the same system behavior as if they were running on the archiver SMP. Achieving the goal of transparency requires that the filesystem API is upheld, preventing the need for changes to applications. Transparency also implies that each cache node will operate within the same consistent namespace and have access to the most recent data.

2.1.4. Cache Nodes Must Act Cooperatively

The primary function of SLASH is to provide read caching support to the archiver SMP to minimize the number tape mounts incurred during everyday use. The most effective means of accomplishing this is to allow caching nodes access to each other's storage directly.

2.1.5. Interface with High Speed Interconnects

One method of optimizing data archiving is to place archival interfaces on the supercomputer's computational interconnect. To do this device drivers must be available for these archival systems. Most SMPs that operate tertiary storage systems do not support interconnects commonly found within HPC systems.

2.1.6. Single Metadata Image

Minimizing overhead is an explicit design goal for this system. Overhead often comes in the form of extra metadata management associated with the state and location of files within the cache. While some solutions employ databases for metadata management, SLASH was designed without a need for databases, keeping configuration and operational overhead low and minimizing the metadata footprint.

3. Architecture

SLASH is based on a modular design that provides abstractions allowing for distribution of HSM duties across several nodes. The purpose of distribution is to shift the disk and CPU load from the archiver SMP to less expensive SLASH nodes. The archiver is still the sole authority for offline data.

SLASH consists of three components: one or more metadata servers, one or more data caching nodes and an archiving node. The metadata service, SLMD, has three primary duties: maintaining and exporting the logical namespace, tracking cache status of files, and lock management. The SLASH Cache Node, SLCN, is responsible for handling data transfer applications, out-of-band data migration, and management of its local cache disk. The SLASH Archiver, or SLAR is responsible for maintaining offline data. At PSC, the SLAR is a Silicon Graphics machine running the "Data Migration Facility" (DMF).

3.1. Logical Namespace Extension

SLASH facilitates access to a single namespace from multiple nodes on a network. At this time, SLASH does not have its own metadata export capability, but instead relies on NFS to export the namespace from the SLMD to the SLCNs. This is an appropriate choice because NFS aptly performs the task, decreases development time, and it is already required when the SLCN is operating in "pass-through mode".

Initial versions of SLASH exported the SLAR's namespace directly to the SLCNs, using the NFS mount for namespace operations and as a target for migrating data to the SLAR. More recent development has abstracted the metadata and placed it on an external server. To allow for multiple SLMDs and other flexibilities, we chose to use an object-based scheme [4]. Indexing data with globally unique object identifiers greatly simplifies namespace locking complexities encountered in a system performing data migration into filesystems imported from other nodes. These complexities are largely caused by directory rename operations occurring before migration of data from SLCNs. Separation of data from metadata in combination with an object naming scheme allows namespace changes to occur completely independent of the data management.

The SLMD is responsible for maintaining accurate size, utimes, and permission information. Namespace

query operations, such as "find" or "ls", in general, should not need to communicate with the SLCNs or SLAR to obtain system metadata information. Only in the case where newly uploaded data has not been migrated will the SLCN be consulted for system metadata information.

3.2. Metadata Management

Metadata internal to SLASH is stored in extended metadata attributes within the metafile's inode. This eliminates the need for a separate database for storing SLASH state. It would be possible to write other data into the metafile but at this point, we have no reason to do so.

The metadata naming scheme requires that files on the SLMD can be directly accessed via their object identifier, or FID, without traversing the user visible namespace. This requirement exists because the SLCNs are not aware of the file's real path. Pushing namespace state to the SLCNs causes severe complications when renaming files and directories, especially when the SLCN holds unmigrated data associated with the renamed location. Providing direct access to data via the FID is accomplished by hardlinking the SLMD "real" object to one named after the FID. Similar to accessing a file via its inode number, this simple technique allows certain SLASH activities to access the metadata without knowledge of the file's path.

SLASH uses three of its own metadata items and one system metadata item (modification time) to maintain the coherency of the data system. Metadata is updated via a simple remote procedure call system comprised of client libraries on the SLCN and a server which runs on the SLMD. The SLMD uses its own lock manager to verify the atomicity of SLASH metadata modifications.

SLASH adds the following metadata items:

3.2.1. Object Identifier or "FID"

The FID is a string comprised of date information, filesystem identification tag, and inode number. This combination of items should guarantee unique identifier creation with the SLASH system.

3.2.2. SLCN Identifier Array

This attribute is the list of SLCN_IDs which hold valid cached copies of the respective object.

3.2.3. Migrated Tag

The migrated tag informs the system of the migration status of the file. When set to false, it signifies that the file has not been archived and only exists on the SLCN.

3.3. Client System Call Intercept Library

All external requests into the namespace are handled by applications running on the SLCNs. Applications requesting I/O operations in the namespace are intercepted and redirected through the client library. The following calls are intercepted: open(), truncate(), create(), close(), write(), and lstat(). Calls such as opendir(), readdir(), rename(), etc. are handled by NFS and ignored by the client library. SLASH clients operate by inserting remote procedure calls into these system calls which communicate with the SLMD on behalf of the application.

3.3.1. Open and Create Protocol

"Open for writing" requests that the SLMD issues an open() with the same options specified in the intercepted open call. So, for example, if the application specified (O_CREAT | O_TRUNC), the SLMD performs an open on the specified pathname with the provided options. If the server open is successful, then the file's FID is created and applied to the extended metadata attributes. The SLCN identifier attribute is set to the SLCN_ID of the requesting node. The FID and error code is returned to the SLCN in the reply message. Using the FID as a unique identifier within its local cache namespace, the caching node opens a local file descriptor and returns it to the application. The application behaves as though it is operating in the global namespace, but in fact its activity is being redirected to the local filesystem.

"Open for reading" operations, such as open() with O_RDONLY, are also intercepted so that cache status can first be determined. Again, the SLASH library inserts an RPC which queries the SLMD for the cache location and FID. The same operation occurs on behalf of lstat(). All of the caching nodes function as entities in a cooperative caching scheme. So if data is cached and accessible, it is read and served directly from the local or remote cache. In a situation where the data is not accessible, as in the case of an SLCN failure, the requested data is transparently read from the SLAR. While this may require a tape mount, it frees the cache system from having to maintain long-term replicated data for redundancy and instead uses tertiary storage as a means for redundancy [1]. At this point the file may be re-cached while the requestor proceeds to access the data directly from the SLAR.

Slash Cache Nodes do not modify each other's local caches, "open for reading and writing" operations, those that do not specify O_TRUNC, are not optimal for SLASH. The protocol for these is the same as that for "open for reading". If the SLCN_ID returned by the SLMD is the equals the SLCN_ID of the requesting

SLCN then the write operation may proceed as usual utilizing the local cache disk. Otherwise, the request must pass through directly to the SLAR via NFS and modify the file there. Problems occur when the requested file has not yet been migrated to the SLAR. In this situation the request will block until the migration has occurred. Though this issue would be disastrous for a typical filesystem, the drawbacks within SLASH are minimal because of the "get" and "put" nature of the archiver. Some threaded data applications do use techniques which require modifying writes but these usually follow create/truncate operations, which presents a less problematic case.

3.3.2. Write Protocol

The SLASH library's procedures on behalf of write() only begin after the first write call is issued. Depending on the cache status of the respective file two things can happen. If the file is cached on the SLCN, the slash library notifies the SLMD, instructing it to set the migrated tag to "false", which signifies that the file needs to be migrated. The SLCN creates a pointer to the cached file which will be used in the migration phase. If the file was being accessed directly on the SLAR then the message instructs the SLMD to strike the SLCN Identifier Array – removing any pointers to now invalid cached copies. Should the remote procedure call fail the I/O is aborted before any data is actually written.

3.3.3. Close Protocol

When a close is issued by the application, the SLMD is requested to set the appropriate utimes on the metadata file. In the case of close after write the filesize is set on the SLMD and the SLCN schedules the file for migration.

3.4. File Object Migration

The migration between the SLCNs and the SLAR is very similar to the migration procedure in an archiver where data moves between disk and tape. The fault scenarios are similar as well. The state in which the data exists only on the SLCN is a critical one. Even though this new data is globally available for reading, only one copy of the data exists until the migration is complete. This fault scenario is similar to that of traditional archivers in situations where the archiver filesystem fails before the data is backed up to tape.

The procedure begins when the SLCN requests a file object migration, the request may be either a put or get. A connection is made to the SLAR's migration server. If the SLAR has an available migration token the node with the file object is instructed to stream it. The migrating file is streamed to a temporary location where upon success is renamed to the FID. If the migration is successful, the SLCN instructs the SLMD to mark the file object's extended attributes. An SLCN "put" transfer requires that the migrated tag is set to true. After the SLCN "get" transfer, the SLCN identifier array is modified with the SLCN_ID.

Timely migration of data to and from the SLAR should not impact the SLARs local disks in a way which causes excessive disk thrashing. To deal with this problem a load management system on the SLAR brokers migrations from the SLCNs while taking into account current tape activity.

4. Re-delegation of Archival Tasks

SLASH aims to lower the hardware cost of large archival systems while providing avenues for expansion. Shifting of archival workloads to cheaper satellite nodes is the primary method of refocusing the archival architecture away from the large SMP [2]. By minimizing the workload of the large SMP we hopefully can prolong the amount of time until an upgrade is needed.

We'd like to reach a point where the archiver's only vital role is the management of tertiary storage. We've identified several workload areas which are good candidates for placement on lower cost nodes.

4.1. File Caching

Probably the most obvious task, file caching is handled by the SLCNs instead of the local disk on the SLAR. Shifting file caching to the SLCN provides several advantages which have already been discussed in Section 2. By enabling caching across a set of cooperative machines, the capacity and bandwidth of the cache can be expanded at a much lower cost than by adding disk and network bandwidth to the archiver. The larger cache inevitably lowers the number of tape mounts needed to fulfill read requests.

4.2. User Data Movement Applications

Placing the users' data tasks on the SLCNs is another obvious method of task offloading. This is important because it prevents excessive context switching across the entire system. Load-balancing user requests across a pool of SLCNs should be more efficient than directing all requests to the SLAR, especially if the system is under heavy duress from tape requests. In cases where

encrypted data transfers are used it is preferred that the tasks run on the cheaper SLCN processors.

4.3. Removal of Small Files From the Archiver Managed Namespace

Many archivers are inundated with millions of small files (< 1MB) [3], PSC's archiver is no exception Overwhelming numbers of small files can cause headaches for system administrators because they lead to excessive backup and restore times. The disk subsystems connected to HPC data archivers are designed to handle large sequential operations. Hence, the filesystem blocksizes are generally large, creating an inefficient scenario for small files. Under normal filesystem use, the handling of small files will inevitably lead to filesystem fragmentation. Shifting the management of small files to archival software, such as DMF, can be wasteful due to the database overhead needed to manage offline files.

These files, the bane of the archival systems, can be managed more efficiently if moved from the SLAR's cache disk to other more suitable storage.

4.4. Exporting of Namespace

Moving namespace management to an external node provides several means of load minimization. Since SLASH uses NFS for namespace operations, having the NFS server on a different node means that requests for metadata will not interfere with the sequential I/O operations taking place on the SLAR. In addition, namespace queries can be fulfilled more rapidly since there is less disk contention.

Shifting namespace responsibilities to an external SLMD allows for SLASH to assume management of small files as discussed in Section 4.1.3.

5. SLASH Storage Application – TCSIO

The "Terascale IO" system [7], developed at PSC, is an object-oriented client-server toolkit. TCSIO provides a client middleware library and the presence of I/O daemons that run on each node where disk resources are to be presented. The I/O daemons are conversant in multiple transport protocols, including a reference implementation in TCP/IP and an expansion to the TCSCOMM library [7], which allows for transfers on the Quadrics network fabric. The protocol for connecting to an I/O daemon begins with a TCP/IP socket connection and includes, among other things, a negotiation to the highest performance protocol for bulk data transport. Control data are always transmitted over the TCP/IP connection.

Within the SLASH environment, TCSIO has been tightly integrated with the SLASH client library. These enhancements allow TCSIO to perform SLASH locator RPCs to determine the location of cached files. The cache location information is used in conjunction with 3rd party transfers to create a powerful "redirection" mechanism for handling access to distributed data. Read requests dispatched within TCSIO are seamlessly redirected to the caching node which holds the relevant data.

TCSIO client requests have built-in fail-over and round-robin distribution capabilities for robustness and parallelism. In the case of a node failure, TCSIO contacts another eligible caching node to obtain the requested data.

6. Benchmark Configuration and Results

We present several performance results for the SLASH implementation currently in production at PSC. The production version does not yet use an external metadata server so only data throughput was tested. The TCSIO system was used as the data transfer mechanism and provided read request redirection. This means that cached reads were served by the SLCN holding the data. The hardware used was the quadrics-based supercomputer, lemieux, six SLCNs, and a 12 processor Silicon Graphics O-300 system (SLAR). To limit the number of measured variables, no filesystems were used on lemieux, /dev/null and /dev/zero were used instead. The JFS filesystem was used on the SLCNs and XFS was used on the SLAR.

6.1. SLCN Configuration

The SLCNs vary in system type though all are Xeon based dual processors machines with four independent PCI busses. At least one PCI bus in each SLCN was PCI-X and was used for the quadrics card. The disk subsystems were comprised of multiple 3ware cards or a single dual-channel scsi card. Each SLCN type was not equal in disk or network throughput. Of the six SLCNs used for testing, two were of each type.

6.1.1. Type 1 SLCN

The type 1 SLCN is a dual processor 3.00 GHz 64-bit Xeon processor with hyperthreading support enabled. The machine has two gigabytes of main memory and uses a Supermicro X6DHE-G2 motherboard. The disk controller is a dual-channel LSI Logic card. The disks are comprised of two 8-bay scsi shelves loaded with 7200 rpm parallel ATA disks. The shelves, responsible for data

redundancy, do raid6 parity (6+2) in hardware. Both scsi devices are striped into one logical unit with a peak write bandwidth of 250 MB/sec. The linux kernel version is 2.6.7 with the 'mm' patch series and quadrics support.

6.1.2. Type 2 SLCN

The type 2 SLCN is also a dual processor Xeon but is a 32-bit model running at 2.79 GHz. Hyperthreading support is enabled here as well. Similar to type 1, this machine has two gigabytes of main memory and uses a Supermicro X5DPE-G2 motherboard. The disk controllers are two 3ware 9500, 12-port, serial ATA cards. The controllers perform raid5 in hardware and have a peak write bandwidth of 125MB/sec. Both 3ware devices are striped into one logical unit with a peak write bandwidth of 250 MB/sec. Type 2 nodes use the same kernel as type 1.

6.1.3. Type 3 SLCN

The type 3 SLCN is a dual processor, 32-bit, 2.66 GHz Xeon with disabled hyperthreading. The Supermicro X5DPE-G2 motherboard has two gigabytes of main memory. The four disk controllers are 3ware 8500 series, 8-port, serial ATA cards. The controllers perform raid5 in hardware and have a peak write bandwidth of 75MB/sec. All four 3ware devices are striped into one logical unit with a peak write bandwidth of 300 MB/sec. The kernel is 2.4.21 based on Redhat Enterprise with quadrics support added.

6.2. Test Results

The first benchmark provides a baseline for each type of SLCN. It tests the TCSIO throughput of raw quadrics, write, and read. Figures 1 through 3 illustrate the performance of each SLCN type. Each machine type has unique performance characteristics. The type 1 SLCN is by far the best writer but is not the best reader. The type 2 nodes read very well but do not scale past two write streams. Though the type 3 machine performs well at three and four raw quadrics streams, when data is written or read from the disk the performance greatly decreases. Filesystem throughput tests on the type 3 SLCNs have shown sequential write performance to be greater than 250 MB/sec. The subpar performance may be related to the 2.4.21 kernel or the disabled hyperthreading. The issue requires further investigation.

Demonstrating the effectiveness of SLASH's distributed caching ability, the second set of benchmarks compares five uncached and cached read and write tests. In each case the number of 10GB data streams is doubled until sixteen is reached. At no point did any lemieux node send more than a single data stream to SLASH.

All operations functioned in the way described in Section 3. Cached operations first communicated with the SLMD and then proceeded to direct incoming data to its local disk. The SLMD ensures that this data is in fact part of the logical namespace. Uncached operations were directed into the SLAR's namespace via NFS. Arguments which dispute the performance of the SGI

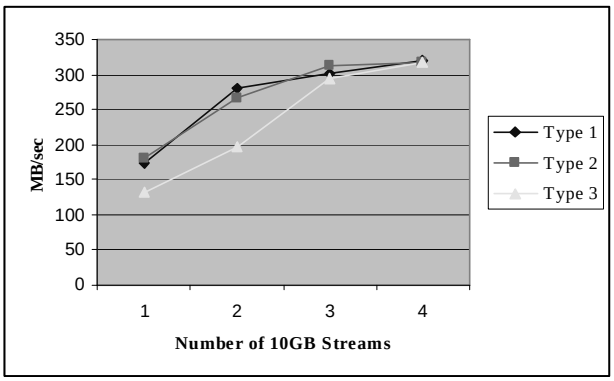

Figure 1. Raw Tcsio / QSW Performance

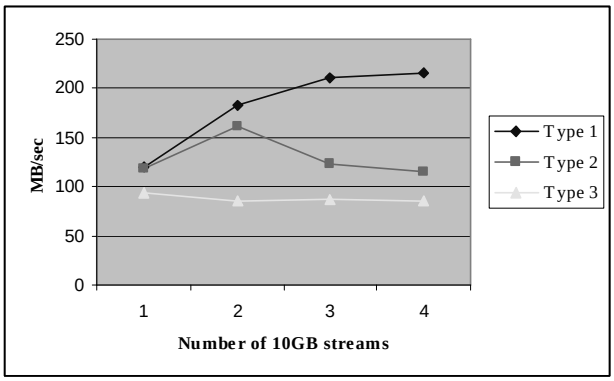

Figure 2. Tcsio/Slash Write Bandwidth

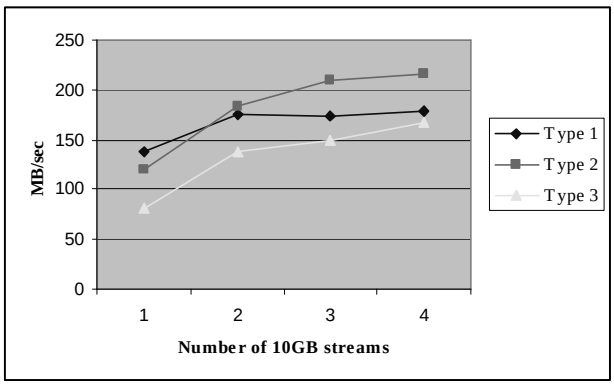

Figure 3. Tcsio/Slash Read Bandwidth

NFS server could be made but we had found each SLCN type to be capable of NFS read and write speeds in excess of 100MB/sec. This is largely due to the excellent asynchronous NFS server implementation created be SGI. The SLCNs and the SLAR are connected by switched gigabit Ethernet.

Figures 4 and 5 contrast cached and non-cached read and writes. The scaling of the SLASH caching system is illustrated here. This test did not aim to show poor SLAR performance but rather to offer a basis of comparison with the cached operations. Though the SLAR has 500MB/sec of disk bandwidth, this was not achieved largely due to the volume group's concatenation scheme – which does not always allow disk operations to exploit all of the disk spindles. At least half of the SLAR's disks were used at any one time.

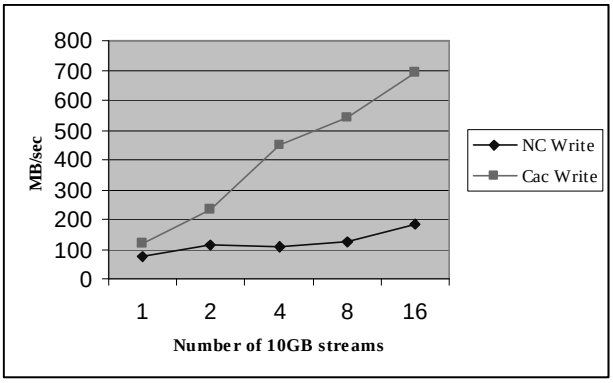

Figure 4. Cached vs. Non-Cached Writes

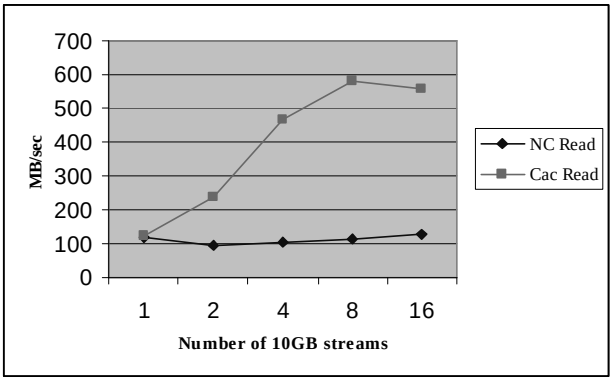

Figure 5. Cached vs. Non-Cached Reads

Because of their mediocre performance, the type 3 SLCNs were not used until the 16 stream test – the only test which used all six cache nodes. Different from the first four tests, the 16 stream uploads used an unbalanced distribution to favor the type 1 SLCNs. While this provided the best scaling it proved harmful during the 16 stream read test. Since the type 1 SLCNs were not superior readers, the overload of data there caused a lower aggregate bandwidth on the read test. A better aggregate read number could have been achieved if the data layout in the cache had favored the type 2 SLCNs.

7. Related Work

The "dCache" [9] system, presented at MSST04 is functionally very similar to SLASH. The creators of dCache have implemented several of the concepts discussed in this paper, such as separation of data from metadata, use of commodity hardware to form a distributed cache, system call intercept libraries, and integration with tertiary storage. DCache is deployed in several locations serving large physics datasets.

Lustre [10] is an object-based parallel filesystems used in several HPC environments. Compared to SLASH, which is designed to function in an HPC hierarchical storage environment, Lustre is a more complex system which is aimed at serving different HPC areas. For performance reasons, most Lustre modules run inside of the kernel, while SLASH is entirely usermode. Lustre's fault tolerance scheme requires storage devices to be shared. Shared configurations likely increase the cost of the system. The external metadata service is a key concept to the flexibility of the filesystem. This idea influenced the thoughts behind the design SLMD. At this time Lustre does not handle offline data or file migration.

The Storage Access Coordination System [5] was designed to optimize the retrieval of high-energy physics datasets from tertiary storage. STACS utilizes sophisticated indexing schemes and request analysis in an attempt to limit the number of tape mounts needed to fulfill pending requests. Systems such as STACS are needed to cope with the multidimensionality of HEP datasets which are being access simultaneously by many users [6]. Most large datasets in an environment such as PSC are checkpoints from large compute jobs which are needed to prime a user's job. Unlike HEP data, in most cases the entire checkpoint dataset needs to be read. In the future when large SLASH requests are fed by the job schedulers we may be able to take advantage of several areas explored by STACS.

8. Future work

More effort needs to be focused on the pre-staging of files into the cache at the beckon of the job schedulers. Techniques explored in [8] could be applied in the archiver realm. This would provide a powerful mechanism for managing the availability of space on the

parallel filesystems along with optimizing tape accesses. Other possibilities provided by object-based storage could be user specified access patterns, correlation of object-identifiers which could be used for pre-caching, or functions such MD5 checksum.

The scalability of the SLMD needs to be analyzed. The design of SLASH allows for the use of multiple SLMDs but this concept must be explored further. If executed properly, the SLMD could be capable of handling billions of files without the use of a database.

9. References

[1] B. Hillyer, A. Silbershatz, "Random I/O Scheduling in Online Tertiary Storage Systems", *Proceedings of the 1996 SIGMOD Conference*, June 1996.

[2] T. Anderson, M. Dahlin, J Neefe, D. Patterson, D. Roselli, R. Wang, "Serverless Network Filesystems", *Proceedings from the SIGOPS '95 Conference*, December 1995.

[3] B. Sarnowska, T. Jones, "Architecture, Implementation, and Deployment of a High Performance High Capacity Resilient Mass Storage Server", *Proceedings from the Eighteenth IEEE Symposium on Mass Storage Systems and Technologies*, April 2001.

[4] M. Mesnier, G. Ganger, E. Reidel, "Object-Based Storage", *IEEE Communications Magazine*, August 2003.

[5] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, A. Sim, "Multidimensional Indexing and Query Coordination for Tertiary Storage Management", *Proceedings from the 11th International Conference on Scientific and Statistical Database Management*, April 1999.

[6] L. M. Bernardo, A. Shoshani, A. Sim, H. Nordberg, "Access Coordination of Tertiary Storage for High Energy Physics Applications", *Proceedings from the IEEE Symposium on Mass Storage Systems*, 2000

[7] N. Stone, J. Kochmar, P. Nowoczynski, J. Scott, D. Simmel, J. Sommerfield, C. Vizino, "Terascale I/O Solutions", *Terascale Performance Analysis Workshop* (2003).

[8] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka, "Informed Prefetching and Caching", *Proceedings of the 15th Symposium of Operating Systems Principles*, December 1995. http://www.pdl.cmu.edu/PDL-FTP/TIP/SOSP15.pdf

[9] P. Fuhrmann, "dCache, the Commodity Cache", *12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies*, April 2004.

[10] Lustre, http://www.lustre.org