Zest

Checkpoint Storage System for Large Supercomputers

Paul Nowoczynski, Nathan Stone, Jared Yanovich, Jason Sommerfield

Pittsburgh Supercomputing Center
Pittsburgh, PA USA
{pauln, stone, yanovich, jasons}@psc.edu

Abstract— The PSC has developed a prototype distributed file system infrastructure that vastly accelerates aggregated write bandwidth on large compute platforms. Write bandwidth, more than read bandwidth, is the dominant bottleneck in HPC I/O scenarios due to writing checkpoint data, visualization data and post-processing (multi-stage) data. We have prototyped a scalable solution that will be directly applicable to future petascale compute platforms having of order 10^6 cores. Our design emphasizes high-efficiency scalability, low-cost commodity components, lightweight software layers, end-to-end parallelism, client-side caching and software parity, and a unique model of load-balancing outgoing I/O onto high-speed intermediate storage followed by asynchronous reconstruction to a 3rd-party parallel file system.

Keywords: Parallel Application Checkpoint, Parallel I/O, Petascale Storage, Client-side Raid, High-performance commodity storage, Terabytes per second, log-structured filesystems.

I. INTRODUCTION

Computational power in modern High Performance Computing (HPC) platforms is rapidly increasing. Moore's Law alone accounts for doubling processing power roughly every 18 months. But a historical analysis of the fastest computing platforms by Top500.org shows a doubling of compute power in HPC systems roughly every 14 months. This accelerated growth trend is due largely to an increase in the number of processing cores; the current fastest computer has roughly 256K cores. An increase in the number of cores imposes two types of burdens on the storage subsystem: larger data volume and more requests. The data volume increases because the physical memory per core is generally kept balanced resulting in a larger aggregate data volume, on the order of petabytes for petascale systems. But more cores also mean more file system clients, more I/O requests to the storage servers and ultimately more seeking of the back-end storage media while storing that data. This will result in higher observed latencies and lower overall I/O performance.

Today, HPC sites implement parallel file systems comprised of an increasing number of distributed storage nodes. However, in the current environment, disk bandwidth performance greatly lags behind that of CPU, memory, and interconnects. This means that as the number of clients continues to increase and outpace the performance improvement trends of storage devices, larger and larger storage systems will be necessary to accommodate the equivalent I/O workload. Through our analysis of prospective multi-terabyte/sec storage architectures we have concluded that maximizing the effective throughput of disks is essential to reeling in the rising cost of large parallel storage systems.

It is common wisdom that disks in large parallel storage systems only expose a portion of their aggregate spindle bandwidth to the application. Optimally, the only bandwidth loss in the storage system would come from redundancy overhead. Today, however, in realistic HPC scenarios the modules used to compose parallel storage systems generally attain < 50% of their aggregate spindle bandwidth. The 'traditional' I/O stack, (Figure 1) illustrates the components of today's storage module. Within the traditional I/O stack several possible culprits may be responsible for performance degradation. Of these, only one need be present: the aggregate spindle bandwidth is greater than the bandwidth of the connecting bus; the raid controller's parity calculation engine output is slower than the connecting bus; and excessive disk seeking. The first two factors are direct functions of the storage controller and may be rectified by matched input and output bandwidths from host to disk.

The final factor, excessive seeking, is a consequence of the filesystem's LBA request ordering being sub-optimal. Large core count environments (on the order of 10^5 - 10^6) have a tendency to exacerbate this problem by presenting large numbers of sequential data steams to the storage system. Generally, parallel filesystems aptly handle sequential data streams. However, we contend that as the stream counts continually increase, the relative performance of traditional I/O modules will continue to decrease because the resulting workloads are becoming more randomized.

Zest attempts to increase the storage sytsem's effective bandwidth through a design which implements performance-aware data placement. Data stored by Zest is done via the fastest mode available to the server without concern to file fragmentation or provisions for storing global metadata. As a result, the current implementation of Zest has

no application-level *read* support. Instead it serves as a transitory cache which copies its data into a full-featured filesystem at a non-critical time. Such a method is well suited for application checkpoint data because immediate readback capabilities are generally not needed.

Figure 1: Traditional Storage System Stack vs. Zest
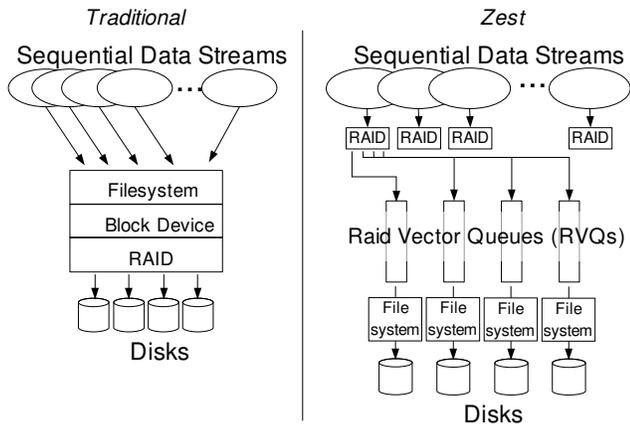
II. Related Work

The Zest design is primarily based on two existing storage technology concepts: log-structured filesystems and object-based parallel filesystems. Design aspects of Zest regarding pull-based I/O were inspired by the concept of *distributed queues* as discussed in [1]. Additionally, Zest's fault tolerance mechanism is somewhat similar to that employed by the commercial parallel filesystem, *PanFS*, developed by Panasas.

A. Log-structured Filesystems

"Recent technology trends suggest that disk input and output may become such a bottleneck in the near future. CPU speeds are increasing dramatically along with memory sizes and speeds, but the speeds of disk drives are barely improving at all. Without new file system techniques, it seems likely that the performance of computers in the 1990's will be limited by the disks they use for file storage." [2]

The above quote, written in 1988, shows that the filesystem community was well aware of the trends facing I/O systems. Surprisingly, these trends still exist today. Over the last 20 years large scale computational systems were able to leverage disk arrays and parallel filesystems. Until today, traditional fileserver components (Figure 1) used in parallel have been able to satisfy the I/O needs of large supercomputers and at a cost relatively low cost. This meant that file layout technologies such as log-structured filesystems were not a critical component to I/O scaling and therefore not aggressively pursued. In fact, log structured systems have

mostly been avoided in parallel I/O systems and have only gained acceptance for use as operation commit logs.

Zest employs a log-structured layout with provisions for a fault-tolerance system which avoid the performance inhibiting characteristics of the traditional storage stack. To accomplish this, some level of divergence from log-structured methods is required. In contrast to a log-structured system, Zest uses a fixed record format which is congruent to the data and parity buffers issued by the client. We define a 'buffer' as a set of data or parity bytes sized to the length of the data portion of a Zest on-disk record. Similar to log-structured systems, Zest places a footer at the end of the buffer which contains the metadata. Here are stored the inode number and the io vectors which describe location of the buffer contents with respect to the file. Combined, the buffer and metadata footer comprise a Zest block. Despite the fact that Zest does not support extents of arbitrary length, the storage performance of Zest is similar to log-structured systems because it allows for any Zest block to be stored in any record container. The result is that streaming disk performance is simple to achieve.

Existing within a checkpoint oriented environment is beneficial for both Zest and other log-structured systems. HPC checkpointing is an inherently burst driven process which is followed by periods of little or no activity. The latent period allows time for segment cleaning activities. Segment cleaning within Zest is referred to as 'syncing'.

B. Object-based Parallel Filesystems

Parallel filesystems provide the infrastructure for today's checkpointing systems. The general architecture for a parallel filesystem consists of a set of traditional I/O nodes serving as object storage devices and a metadata server. Applications achieve I/O parallelism by striping data across the set of I/O nodes in a manner equivalent to RAID0. This method has been widely successful due to its straightforward manner of parallelization and its efficient metadata layer.

The strength of object-based parallel filesystem metadata lies in its ability to index arbitrary amount of data through a small fixed size map. The map data structure is composed merely of an ordered list of storage servers and a stride. In essence, the map describes the location of the file's sub-files and the number of bytes which may be accessed before proceeding to the next subfile or stripe. Besides the obvious advantages in the area of metadata storage, there are several caveats of this method. The most obvious is that the sub-files are the static products of the object metadata model which was designed with its own efficiency in mind. The result is a deterministic data placement method which requires data to be sent to a specific I/O node and placed within a predetermined container file at a fixed location.

With the exception of *PanFS* [3]*,* today's parallel file servers rely on server-based disk fault tolerance provided by a RAID'd set of disks (Figure 1). The raid layer further complicates matters by incorporating several spindles into the

same block device address range and forcing them to be managed in strict unison. Through this strict binding the RAID set is susceptible to performance degradation when one or more disks are perturbed. To make matters worse, because of striping the entire parallel storage system is subject to degradation when a single I/O server is slow. In effect, a single slow disk or rebuilding RAID set can negatively impact an entire storage cluster.

Zest is similar to today's parallel filesystems in that it allows applications to store data in parallel to a set of I/O servers. Zest, however, was designed to maximize disk throughput and therefore does not rely on server-based RAID, centrally mandated data placement, or access to a single disk by multiple threads. These characteristics starkly contrast those of today's parallel I/O systems.

III. DESIGN CONCEPTS

A. Non-Deterministic Data Placement

Zest is designed to perform sequential I/O whenever possible. To achieve a high degree of write request sequentiality to each disk, Zest's block allocation scheme is neither determined by the file object identifier, the extent information, or a RAID sub-system but rather the next available Zest block on the disk. More importantly, the sequentiality of the allocation scheme is not affected by the number of clients, the degree of randomization within the incoming data streams, or the RAID attributes (i.e. parity position) of the block. Because it minimizes seeks, this simple, non-deterministic data placement method is extremely effective for presenting sequential data streams to the spindle.

B. Client-side Parity Calculation

In order to prevent potential server-side raid bottlenecks, Zest places the parity generation and checksumming workload onto the clients. The HPC resource, which is the source of the I/O, has orders of magnitude more memory bandwidth and CPU cycles at its disposal than that of the storage servers. Placing the parity workload onto the client CPUs saves the storage system from requiring costly raid controllers and guarantees that parity generation will not impede performance. In addition to Zest, other parallel filesystems such as the commercial filesystem *PanFS* [X] by Panasas employ client-side parity generation.

C. No Leased Locks

To minimize network RPC overhead; features which induce blocking; and the complexity of the IO servers; Zest purposely does not use leased locks. Instead, it ensures the integrity of intra-page, unaligned writes performed by multiple clients. Typically, filesystem caches are page-based and therefore a global lock is needed to ensure the update atomicity of a page. Zest does not use such a method; instead it uses vector-based write buffers. One possible caveat of this method is that Zest cannot guarantee transactional ordering for overlapping writes. Since it is uncommon for large parallel HPC applications to write into overlapping file offsets we do not feel that this is a fatal drawback.

IV. SERVER DESIGN

The Zest I/O server appears as a storage controller and file server hybrid. Similar to a controller, it manages I/O to each drive as a separate device. Disks are not grouped into volumes or multi-device LUNs. In the vein of a file server, the Zest server is aware of file inodes and extents. The combination of behaviors enables Zest to interact with a filesystem in a way which does not inhibit performance. The Zest I/O server is composed of several subsystems which are described here.

A. Networking and RPC Stack

Zest uses a modified version of the *LNET* and *ptlrpc* libraries found in the *Lustre* filesystem. Following our evaluation of the *Lustre* rpc library it was decided to adopt the implementation because of its proven robustness, performance, and logical integration with the *LNET/Portals* API. *Ptlrpc* also provides a service layer abstraction which aids in the creation of multi-threaded network servers. Zest makes use of this service layer to establish two RPC service for I/O and metadata. The Zest I/O and metadata services are groups of symmetric threads which process all client RPCs. Metadata RPCs are not concerned with bulk data movement but instead interface with the Zest server's inode cache and the namespace of the accompanying full-featured filesystem. The I/O service is responsible for pulling data buffers from the clients and passing them into the write fifo's called *raid vectors queues*.

B. Disk I/O Subsystem

The Zest disk I/O subsystem assigns one thread for each valid disk as determined by the configuration system. Disk numbers are assigned at format time and are stored within the Zest superblock. Each disk thread is the sole authority for his disk, it duties include: performing reads and writes, io request scheduling, rebuilding active data lost due to disk failure, free space management and block allocation, tracking of bad blocks, and statistics keeping.

The disk I/O system consumes Zest blocks from a set of queues called *raid vectors queues (RVQ)*. An RVQ is a single fifo used to pass Zest blocks from the Rpc stack to the disk threads. The Rpc stack places each member of an incoming parity group onto its corresponding RVQ. A parity group is a set of Zest blocks (N data, 1 parity) which have been XOR'd. The RVQ construct ensures that no two buffers of a single parity group are stored onto the same disk. Disks are bound to an RVQ based on the number assigned to them at format. Given a 16-disk Zest server, a 3+1 RAID scheme would create four RVQs where disks [0-3] were assigned to queue0, disks [4-7] to queue1, and so on. Such a configuration allows for multiple drives to process write requests from a single queue resulting in a pull-based I/O

system where incoming requests are handled by the devices which are ready to accept them. Devices which are slow naturally take less work and devices recognized as failed, remove themselves from all raid vector queues. This technique is similar to that of *distributed queues* presented in [1]. In order to be present on multiple raid vectors, disk I/O threads have the ability to simultaneously block on multiple input sources. This capability allows for each disk thread to accept write I/O requests on behalf of many raid schemes and read requests from the *syncer* and *parity regeneration* subsystems (described below).

C. Syncer and File Reconstruction

Presently, Zest does not support globally stored file metadata therefore it relies on copying its data into a full-featured filesystem to present the data for readback. In practice this accompanying filesystem exists on the same physical storage and the copy process occurs after the checkpoint process has completed.

Upon storing an entire parity group from a client, the completed parity group is passed into the syncer's work queue. From there the syncer issues a read request to each disk holding a member of the parity group. The disk I/O thread services this read request once all of the write queues are empty. Once the read I/O is completed, the read request handle is passed back the syncer. From there it is written to the full-feature filesystem via a pwrite syscall (Note: the io vector parameters necessary for the system pwrite were stored in the Zest block footer). When the entire parity group has been copied out, the syncer instructs the disk threads to schedule reclamation for each of the synced blocks. Reclamation occurs only after all members of the parity group have been copied out.

The syncer is required to perform a checksum on the data returned from the disk. This checksum protects the data and it associated metadata. In the event of a checksum failure, the block is scheduled to be rebuilt through the *parity regeneration* service.

D. Strong Parity Declustering and Regeneration**n**

Zest's parity system is responsible for two primary tasks: storing declustered parity state and the reconstruction of failed blocks. Prior to being passed to the *syncer* process, completed parity groups are handed to the parity declustering service. Here the disk identifier and block number pairs of the parity group members are collated and are stored to a block device called the parity device. The set of pairs comprising the entire parity group are called a parity group descriptor (PGD). Each PGD is written to N times, once for each parity group member at an address unique to the member's disk and block number. Indexing the *parity device* by disk and block number allows for inquiry on behalf of corrupt blocks where the only known information are the disk and block numbers. This is necessary for handling the case of a corrupt Zest block.

During normal operation, the *parity device* is being updated in conjunction with incoming writes in an asynchronous manner by the *parity device* thread. Operation is purposely asynchronous to minimize blocking in the disk I/O thread's main routine. As a result, the parity device is not the absolute authority on parity group state. Instead, the on-disk structures have precedence in determining the state of the parity groups. Currently at boot time, a group finding operation joins active parity groups and the *parity device* is verified against this collection. In the event of a failed disk, the parity device is relied upon as the authority for the failed disk's blocks. In the future this fsck-like operation will be supplemented with a journal.

V. Client Design and Implementation

The Zest client currently exists as both a *FUSE* (file system in user-space) mount and a statically linkable library. The latter, designed primarily for the *Cray XT3*, is similar to the *liblustre* library in that it is single-threaded. It should be recognized that the Zest system is not optimized for single client performance but rather large multitudes of parallel clients. Therefore it stands to reason that despite the Zest server being equally accessible by all compute processors, the zest client does not stripe its output across Zest servers. Instead, the group of client processors are evenly distributed across the set of Zest servers. We expect to make use of this behaviour to implement checkpoint bandwidth provisioning for mixed workloads.

A. Parity and Checksum Calculation

As described above, the Zest client is tasked with calculating parity on its outgoing data stream and performing a 64-bit checksum on each write buffer and associated metadata. Performing these calculation on the client distributes this workload across a larger number of cores and optimizes the compute resource performance as a whole by allowing the Zest servers to focus on data throughput.

The Zest parity system is configurable by the client based on the degree of protection sought by the application and the hardware located at the server. At present, Zest gracefully handles only single device failures through RAID5. Both Zest server and client are equipped to handle short write streams where the number of buffers in the stream is smaller than the requested raid scheme.

B. Write Aggregation

The client aggregates small I/Os into its vector-based cache buffers on a per file-descriptor basis. Designed to work on an MPP machine such as the *Cray XT3,* Zest assigns a small number of data buffers to each file descriptor. These buffers can hold any offset within the respective file though a maximum number of fragments (vectors) per buffer is enforced. This maximum is determined at Zest server format time and is directly contingent on the number of io vectors which can be stored in the metadata region of a Zest block.

Typically we have configured this maximum to 16 meaning that a client may fill a write buffer until either its capacity is consumed or the maximum number of fragments has been reached.

C. Client to Server Data Transfer

The elemental transfer mode from client to server is pull-based, implemented via *LNetGet()*. As Zest blocks are consumed, they are scheduled to be retrieved by the Zest server. Ensuring the viability of the client's parity groups requires the client to hold the entire parity group until all of its respective Zest blocks have been acknowledged by the Zest server. Zest supports both write-back and write-through server caching, the protocol for acknowledgement hinges around the caching policy requested by the client. Depending on the size of the *write()* request and the availability of buffers, the client may use zero-copy or buffer-copy mode.

One pivotal advantage of *non-deterministic data placement* is that it allows Zest clients to send parity groups to any Zest server within the storage network. The result is that, in the event of a server failure, a client may resend an entire parity group to any other Zest server. We predict that this feature will be extremely valuable in large parallel storage networks because it allows for perfect rebalancing of I/O workloads in the event of server node failures and therefore eliminates the creation of hot spots.

VI. ZEST PERFORMANCE RESULTS

Here are preliminary performance results from a single 12-disk Zest server and 3 client nodes (each with 8 xeon cores). The Zest server's drives are SATA-2 and operate at a sustained rate of 75MB/s. The Zest server self-test, which tests only the disk I/O code path, measured a sustained rate of 868MB/s (96.4% of aggregate max sustained rate). This measurement is labelled 'ST' in Figure 2.

Figure 2

Zest Server Performance
Storing Large Sequential Streams

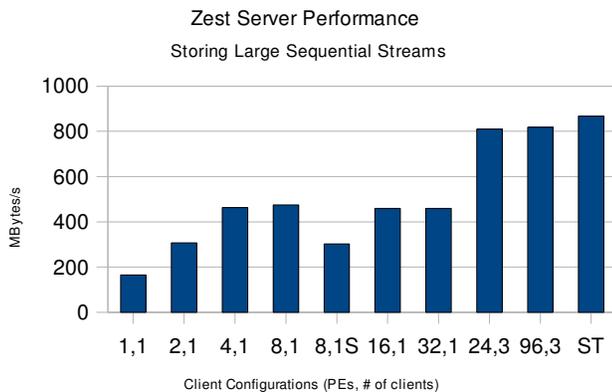Client Configurations (PEs, # of clients)

Figure 2 shows that in the best cases (using 3 clients) the end-to-end throughput of the Zest server hovers around 90% of its aggregate spindle bandwidth. The test application, using a RAID5 7+1 which incurs a 12.5% overhead, saw 80% of the aggregate (720MB/s). These tests provide good insight into the performance of the client and server. The Fuse-based (Filesystem in Userspace) client shows relatively good performance though there is a ceiling around 480MB/s. The datapoint 8,1S indicates that the ceiling is most likely a memory bandwidth issue as the other tests were run on a 2.67Ghz xeon (with a faster memory bus). It should be noted that the Fuse version used did enforce a 128KB maximum I/O size. We expect that once this limitation is removed (linux 2.6.26) the client rates should increase.

VII. CONCLUSION AND FUTURE DEVELOPMENT

Zest is designed to facilitate fast parallel I/O for the largest production systems currently conceived (petascale) and it includes features like configurable client-side parity calculation, multi-level checksums, and implicit load balancing within the server. It requires no hardware RAID controllers, and is capable of using lower cost commodity components (disk shelves filled with SATA drives). We minimize the impact of many client connections, many I/O requests and many file system seeks upon the backend disk performance and in so doing leverage the performance strengths of each layer of the subsystem.

Much contemplation has been given to the development of a global-metadata system for Zest. At this time (Fall '08), the high-level design has been considered and at least one key data structure has been implemented for this purpose. It is most likely, however, that immediate efforts will be put forth into stabilization and performance improvements of the present transitory caching system.

REFERENCES

[1] Remzi H. Arpaci-Dusseau , Eric Anderson , Noah Treuhaft , David E. Culler , Joseph M. Hellerstein , David Patterson , Kathy Yelick, Cluster I/O with River: making the fast case common, Proceedings of the sixth workshop on I/O in parallel and distributed systems, p.10-22, May 05-05, 1999, Atlanta, Georgia, United States.

[2] Mendel Rosenblum , John K. Ousterhout, The design and implementation of a log-structured file system, ACM Transactions on Computer Systems (TOCS), v.10 n.1, p.26-52, Feb. 1992.

[3] *www.panasas.com/**panfs**.html*