Zest: The Maximum Reliable Terabytes Per Second Storage for Petascale Systems

Paul Nowoczynski, Nathan Stone, Jared Yanovich, Jason Sommerfield

Pittsburgh Supercomputing Center

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 on the Cray XT3 compute platform 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.

1. 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, so that the first petaflop computing platform is expected in late 2008. 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 increasing the bandwidth efficiency of constituent disks is essential to reeling in the rising cost of large parallel storage systems and minimizing the number of storage system components.

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. There are several possible culprits which may be responsible for this degradation, of them, only one need be present to negatively impact performance: 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 sub-optimal LBA request ordering caused by the filesystem. 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 last factor, which is essentally 'seek' overhead, is more difficult to overcome because of the codependence of the disk layer and filesystem on the simple linear block interface. 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.

Zest attempts to increase per-spindle efficiency through a design which implements performance-wise data placement as opposed to those which are more friendly to today's filesystem metadata schemas. 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. This method is well suited for application checkpoint data because immediate readback capabiliies are generally not needed.

2. Design Concepts

The Zest checkpoint I/O system employs three primary concepts to achieve its performance target of 90% aggregate spindle bandwidth.

2a. Relatively Non-Deterministic Data Placement

Zest is designed to perform sequential I/O whenever possible. To achieve a high degree of sequentiality, Zest's block allocation scheme is not determined by data offset or the file object identifier but rather the next available block on the disk. Additionally, 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. It should be noted that a block's parity position does restrict the number of disks which may handle it. This is the only determinism maintained in the write process and is necessary to uphold the semantics of the Raid scheme.

Prior to the advent of petascale computing this data storage method would be considered prohibitive because it destroys two inferential systems which are critical to today's parallel I/O infrastructures: the object-based parallel file system metadata schema and the block-level RAID parity group association. RAID systems infer that every same numbered block within the respective set of spindles are bound together to form a protected unit. This method is effective because only the address of a failed block is needed to determine the location of its protection unit 'cohorts' with no further state being stored. Despite this inferential advantage, we contend that strict parity clustering can be detrimental to performance because it pushes data to specific regions on specific disks.

Object-based parallel file systems use file-object maps to describe the location of a file's data. These maps are key components to the efficiency of the object-storage method because they allow for arbitrary amounts of data to be indexed by a very small data structure 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 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 an overly deterministic data placement method in which by forcing I/O into a specific sub-file, increases complexity at the spindle because the backing filesystem's block allocation schemes cannot guarantee sequentiality in the face of thousands or millions of simultaneous IO streams.

2b. 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.

2c. 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.

3. Server Design

The Zest I/O server, otherwise known as a *zestion*, appears as a storage controller / file server hybrid. Similar to a controller, the *zestion* manages I/O to each drive as a seperate device. I/O is not done into a virtual lun of multiple disks or volumes but rather to each disk. In the vein of a file server, the *zestion* is aware of file inodes and file extents. This combination of behaviors enables Zest to interact with a filesystem in a way which does not inhibit performance.

The *zestion* is composed of several subsystems which are described here.

3a. Networking and RPC Stack

Zest uses a modified version of the *LNET* and *ptlrpc* libraries found in the *Lustre* filesystem. There are several reasons for this, the primary being the need to maintain capability with *LNET* routers for use on the *Cray XT3.* Presently, Zest supports both usermode *LNET* drivers (tcplnd and uptlld). On the *zestion,* the *tcplnd* is the functional equivalent to kernel mode Lustre *ksocklnd.* Some modifications were made to *tcplnd* for supporting multi-rail configurations, per-interface statistics, and the enabling of server-mode.

After further investigation into 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 services: IO and metadata. The Zest IO 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 *zestion's* inode cache and with the namespace of the accompanying full-featured filesystems. The IO service is responsible for pulling data buffers from the clients and passing them into the write processing queues called *raid vectors*.

3b. Disk I/O Subsystem

The Zest disk I/O subsytem 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, freespace management and block allocation, tracking of bad blocks, and statistics keeping.

In order to ensure proper RAID semantics, the disk I/O system interacts with a set of queues called *raid vectors.* This construct exists to ensure that write blocks of differing parity positions are not stored onto the same disk. *Raid vectors* are filled with write buffers by the Rpc stack, who appropriately places incoming buffers into their respective queues. Disks are assigned to raid vectors based on their number. Given a 16-disk zestion, a 3+1 RAID scheme would create four *raid vector* queues where disks[0-3] were assigned to queue0, disks[4-7] to queue1, and so on. This configuration allows for multiple drives to process write requests from a single queue. The result of this design is a pull-based I/O system where incoming I/O's 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. 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).

The disk I/O subsystem may be configured to use one of three access modes: scsi generic, block, or file. The scsi generic mode provides a zero-copy I/O path to the disk, we have found this mode to be extremely efficient. The 'file' mode is useful for testing and debugging on workstations which do not have multiple disk drives available for Zest use.

3c. Syncer and File Reconstruction

At present, 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 checkpoing process has completed.

Upon storing an entire parity group stream 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 (the io vector parameters necessary for the system pwrite were provided by the client and stored adjacently to the file data). When the entire parity group has been copied out, the syncer instructs the disk threads to schedule reclamation for each of the synced blocks. This process occurs only after all members of the parity group have been copied out.

The syncer, and other Zest 'read' clients, are required to perform a checksum on the data returned from the disk. This checksum protects the data and it associated metadata (primarily the describing io vectors). In the event of a checksum failure, the block is scheduled to be rebuilt through the *parity regeneration* service.

3d. Parity Declustering and Regeneration

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 where they are stored to a solid-state device *(parity device)*. *Parity device* addressing is based on the disk and block numbers of the newly written blocks. 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. The parity group structure is a few hundred bytes in size and lists all members of the protection unit. For each member in the parity group, the structure is copied to that member's respective *parity device* address.

During normal operation, the *parity device* is being updated in conjunction with incoming writes in an asynchronous manner by the *parity device* thread. This 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 declustered parity groups. Currently at boot time, active parity groups are joined by a group finding operation 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.

4. 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, used 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 *zestions* being equally accessible by all compute processors, the zest client does not stripe its output across *zestions.* Instead, the group of client processors are evenly distributed across the set of *zestions.* We expect to make use of this behavior to implement checkpoint bandwidth provisioning for mixed workloads.

4a. 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 *zestions* 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.

4b. 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 *zestion* format time and is directly contigent 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.

4c. Client to Server Data Transfer

The elemental transfer mode from client to server is pull-based, implemented via *LNetGet()*. As write buffers are consumed, they are placed into a *rpc set* and the *zestion* is instructed to schedule the retrieval of the buffer. The *rpc set* is a functional construct of the *lustre ptlrpc* library which allows groups of semantically related rpc requests to be managed as a single operation. This fits nicely with Zest's concept of parity groups, hence, the zest client assigns an *rpc set* to each active parity group.

Ensuring the viability of the client's parity groups requires the client to hold it buffers until the entire group (or *rpc set*) has been acknowledged by the *zestion*. 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, zero-copy or buffer-copy mode may be used.

One pivotal advantage of *relative non-deterministic data placement* is that it allows Zest clients to send parity groups to any *zestion* 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 *zestion*. 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.

5. Server Fault Handling

5a. Media Error Handling

All Zest data and metadata are protected by a 64-bit checksum used to detect media errors. On 'read' the *zestion* verifies the checksum to ensure that the block has not been compromised. When a bad block is found its parity group information is located via a lookup into a separate device. The *parity device* is a solid state memory device whose purpose is to maintain the parity group structure for every block on the system. Any Zest block's parity group descriptor is located via a unique address composed of the block's disk and block identifiers. Since the IO pattern to the parity device is essentially random it has been outfitted with a small solid-state disk. Currently an 8 million block Zest system requires a 4 gigabyte parity device. The parity device update path is asynchronous and therefore, if needed, the entire device may be reconstructed during file system check.

5b. Run-time Disk Failures

Since the Zest system manages RAID and file-objects, handling of disk failures only requires that volatile blocks are rebuilt. Zest has full knowledge of the disk's contents so old or unused blocks are not considered for rebuilding. During the course of normal operation, Zest maintains lists and indexes of all blocks being used within the system. In the event of a disk failure, the set of blocks who have not been synchronized or whose parity group cohorts have not been synchronized will be rebuilt. Here the parity device will be used to determine the failed block's parity group cohorts. It must be noted that, at this time, Zest cannot recover from simultaneous failure of a parity device and a data disk.

5c. File System Check and Boot-time Failures

On boot, the file system check analyzes the system's parity groups and schedules the synchronization of volatile blocks. If a disk fails in the start-up phase any volatile blocks which it may have been holding are located during the fsck process and rebuilt.

5c. Multi-pathing and Failover

Zest servers support pairwise dynamic fail-over through disk multi-pathing, disk UUID identification, and Linux-HA software. *Zestion* fail-over pairs are configured to recognize each others disks through the global configuration file. Since zest clients are able to resubmit writes to other *zestions*, the fail-over procedure is not as fragile or imminent as one might expect. The primary post fail-over activity is to examine the partner's disks in search of non-synchronized data and process that data through the syncer.

6. Zest within the Cray XT3 Environment

Adapting the *Cray XT3 I/O* environment, the Zest server cluster is located on an InfiniBand cloud external to the machine. The *Cray SIO* nodes act as a gateway between the *XT3's* internal network and the Zest servers. *Lustre LNET* routing services are run on each of the SIO nodes to route traffic from *Seastar* compute interconnect to the external InfiniBand network, providing seamless connectivity from compute processors to the external Zest storage and services.

To achieve compatibility with the Lustre routing service, Zest's networking library is largely based on Lustre's LNET and RPC subsystems. The Zest server uses a modified TCP-based Lustre networking driver with additional support for InfiniBand sockets-direct protocol. Since Zest is a user mode service, it does not have access to the native, kernel-mode InfiniBand Lustre driver so SDP was chosen for its convenient path into the InfiniBand interface.

7. Zest Performance Results

Here are preliminary performance results from a single 12-disk *zestion.* Both the PSC Cray XT3 and a small linux cluster we used as client systems. The *zestion's* drives are SATA-2 and operate at a sustained rate of 75MB/s. The maximum observed performance of the entire set of disks, tested with scsi generic io, was 900MB/s. The *zestion self-test*, which utilizes the I/O codepath, without using the rpc layer, measured a sustained back-end bandwith rate of 840MB/s.

To date, the best numbers are attained when using the linux cluster as a client. This is due to the use of the Infiniband sockets-direct protocol as the transport. We have not been successful running the *Lustre ksocklnd* with sockets-direct so tests performed from the Cray XT3 have relied on IP over Infiniband (IPoIB). The IPoIB path is significantly inferior to the sockets-direct protocol. It should be noted that these tests did not utilize zero-copy socket-direct protocol.

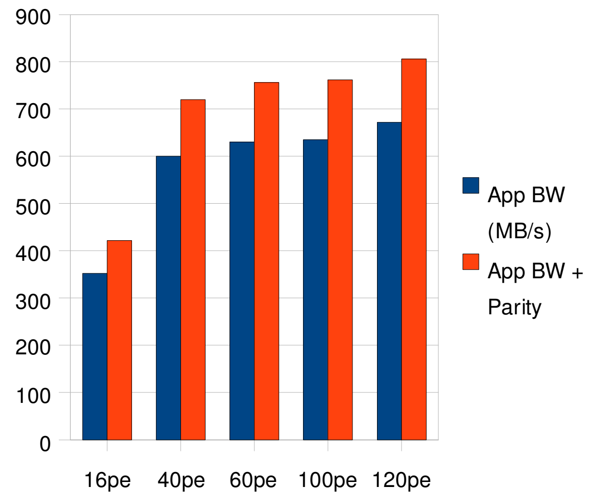
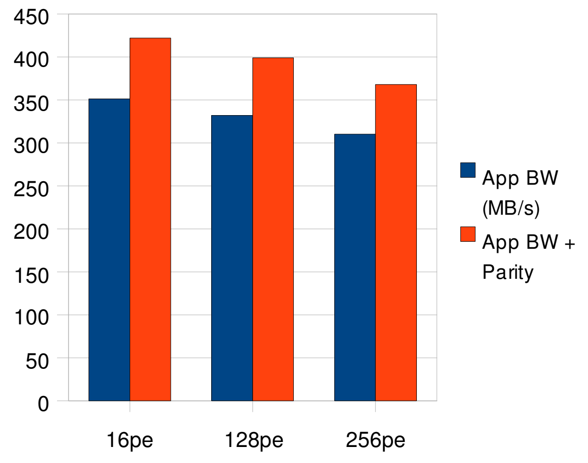


Figure P0: Linux Cluster: Sockets-direct protocol, Client Raid 5+1. (Y-Axis is MB/s)

Figure *P0* shows that in the best case, 120 client threads, the end-to-end throughput of the *zestion* was 89.6% of the aggregate spindle bandwidth. However the application, using a RAID5 5+1, which incurs a 17% overhead, saw 75% of the aggregate. It can be safely extrapolated that had the application used an 11+1 parity stripe, ~82% of the aggregate would have been realized. This result shows that more work must be done to eliminate the 8% loss to server overhead. It also demonstrates that the application observed bandwidth is largely dependent on the raid scheme used by the client.



Firgure P1: Cray XT3, IPoIB protocol, Client Raid 5+1. (Y-axis is MB/s)

Figure *P1*shows Zest performance when using the Cray XT3 as a client. This XT3 client test shows that IpoIB is not an effective interconnect. We are assured that IPoIB is the culprit because equally poor results were observed while using this mode on the linux cluster (280MB/s from 100pe's). As described above in Section 6, the test used several *Cray SIO* as *Lnet routers.* The routers were configured to use IPoIB since sockets-direct was not available. In an attempt to maximize throughput, the *zestion* employed multiple IpoIB interfaces. These interfaces were joined into the same *Lnet network* through the multi-rail feature which we added into the *Lnet tclpnd*.

10. 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.

In the future we aim to improve network performance by leveraging the upcoming *Lustre LNET* user to kernel mode bridging module. This module provides LNET api compatibility to userspace applications therefore it should be easily adapted by Zest. It is our hope that using the native *Lnet RDMA* drivers will substantially increase efficiency and throughput.

Much contemplation has been given to the development of a global-metadata system for Zest. At this time (Spring '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.

About the Authors

Paul Nowoczynski, Jared Yanovich, Nathan Stone, and Jason Sommerfield are members of the Pittsburgh Supercomputing Center's Advanced Systems Group.