Zest I/O

Paul Nowoczynski, Jared Yanovich

Advanced Systems, Pittsburgh Supercomputing Center

Cray Users Group, Helsinki, May 8th, 2008

What is Zest?

“Parallel storage system designed to accelerate application checkpointing on large compute platforms.”

- Designed to expose 90% of the total disk spindle bandwidth to the application in a reliable fashion.
- “Write-only” intermediate store with no application read capability.
- End-to-end design, from client software down to the individual IO nodes' disks.
- Emphasizes the use of commodity hardware.

Zest - Who & When

- Designed by the PSC Advanced Systems Group (*Nowoczynski, Yanovich, Stone, Sommerfield*).
- Design work began in September '06.
- Initial development took about one year.
- Prototype stabilized in Fall of '07.
  - SC '07 Storage Challenge
- Currently most major features are implemented and in test.

Goal and Motivations

Obtain the highest possible bandwidth for reliable application checkpointing at a relatively low cost. Why?

- Disk drive performance is being out-paced by the other system components which is leading to disproportional IO costs.
- In the largest machines memory capacity has increased about 25x over the last 7 years, while disk speeds have increased about 4x over the period.
- Today's I/O systems do not deliver a high-percentage of spindle bandwidth to the application.

Why Target Checkpointing?

Optimizing checkpoint operations can increase the overall computational efficiency of the machine.

...Time not spent in I/O is probably time 'better spent'.

- Application blocks while checkpointing is in progress.
- Increase in write I/O bandwidth allows to machine to spend more time doing real work.
  - Maximize “compute time / wall time” ratio

Why Target Checkpointing?

Checkpoint operations have characteristics that create interesting opportunities for optimization:

- Generally write dominant and accounts for most of the total I/O load.
  - 'N' checkpoint writes for every 1 read.
- Periodic
  - Heavy bursts followed by long latent periods.
- Data does not need to be immediately available for reading.
- The dominant I/O activity on most HPC resources!

Symbiosis: Zest + Checkpointing

Zest aims to provide the application with extremely fast snapshotting capabilities.

In turn, Zest relies on the characteristics of the CP procedure to do its job effectively.

- No need for application read support
- Time for post-processing snapshotted data between snapshots.

Today's Throughput Problems

Goal is to expose 90% of the spindle bandwidth to the application. What prevents I/O systems from achieving this now?

- Aggregate spindle bandwidth is higher than the bandwidth of the controller or the connecting bus.
- Raid controller (parity calculation) may be a bottleneck.
- Disk seeks

If an I/O system could overcome these performance inhibitors then 90% would be within reach.

Today's Throughput Problems

Of the three primary performance inhibitors, disk seeks are perhaps most difficult to squash.

- By nature, supercomputers have many I/O threads which are simultaneously accessing a backend filesystem on an I/O server.
- For redundancy purposes, backend filesystems are conglomerations of disks. In most cases, a group of drives will be involved in the processing of a single I/O request. Competing requests actually cause entire drive groups to seek.

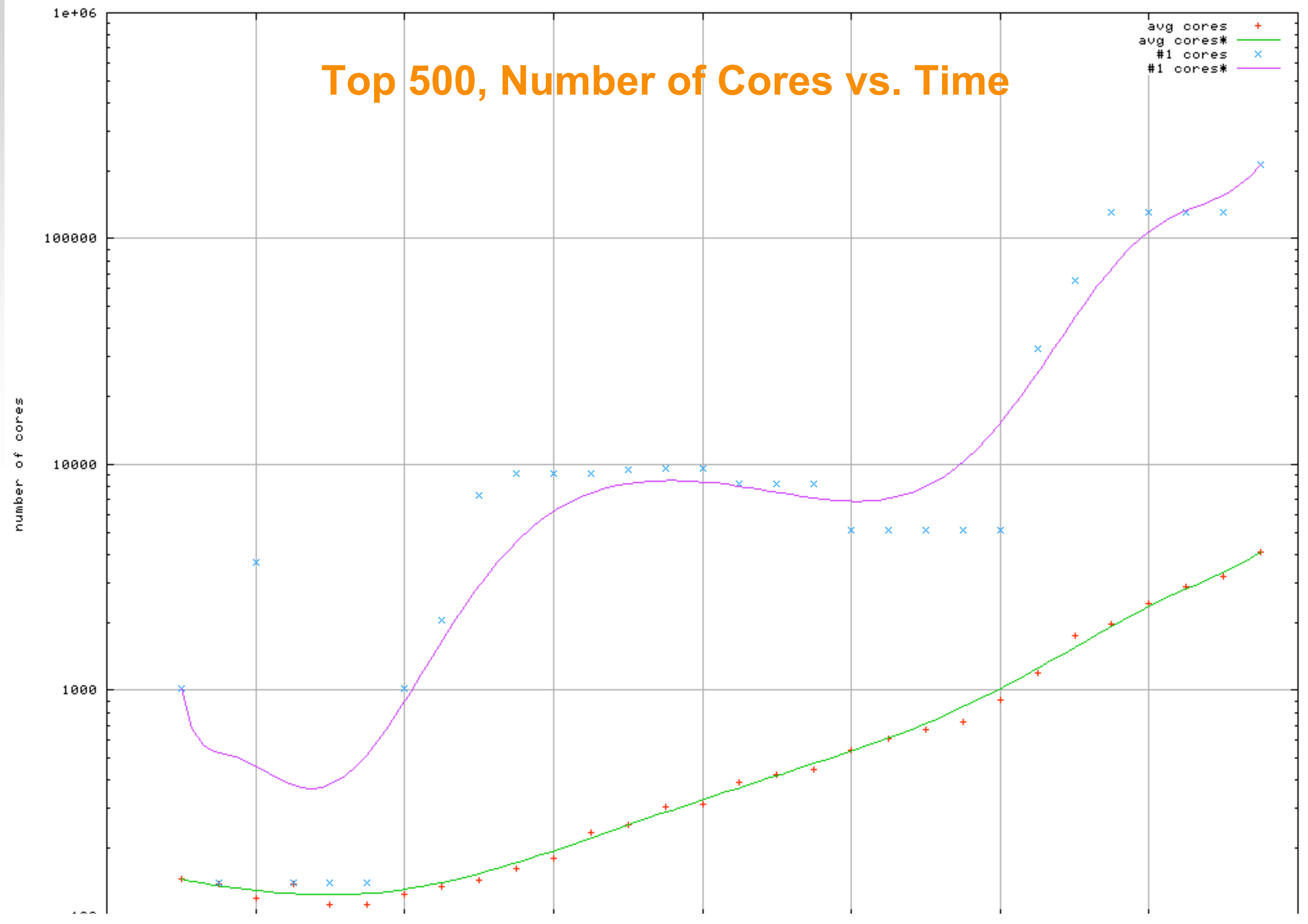
To make matters worse..

Today's Throughput Problems

The evolution of multi-core processors is drastically increasing the number of possible I/O consumers in today's HPC systems.

Increasing core counts exacerbate disk seeks by adding more execution elements into the mix which means:

- Greater number of I/O requests.
- Higher degree of request randomization seen by the I/O server.


Top 500, Number of Cores vs. Time
number of cores of top500 computers
number of cores
avg cores
avg cores*
#1 cores
#1 cores*
1e+06
100000
10000
1000

Zest Write Techniques

Zest uses several methods to minimize seeking and optimize write performance.

- Each disk is controlled by single I/O thread which heavily sequentializes incoming requests.
- Relative, non-deterministic data placement. *(RNDDP)*
- Client generated parity.

Zest Write Techniques

Disk I/O Thread

… fairly straightforward

- Exclusive access prevents thrashing.
- Has a rudimentary scheduler for managing read requests from the syncer threads.
- Maintains a map of free and used blocks and is able to place any data block at any address
- Pulls incoming data blocks from a single or multiple queues called "*Raid Vectors*".

Zest Write Techniques

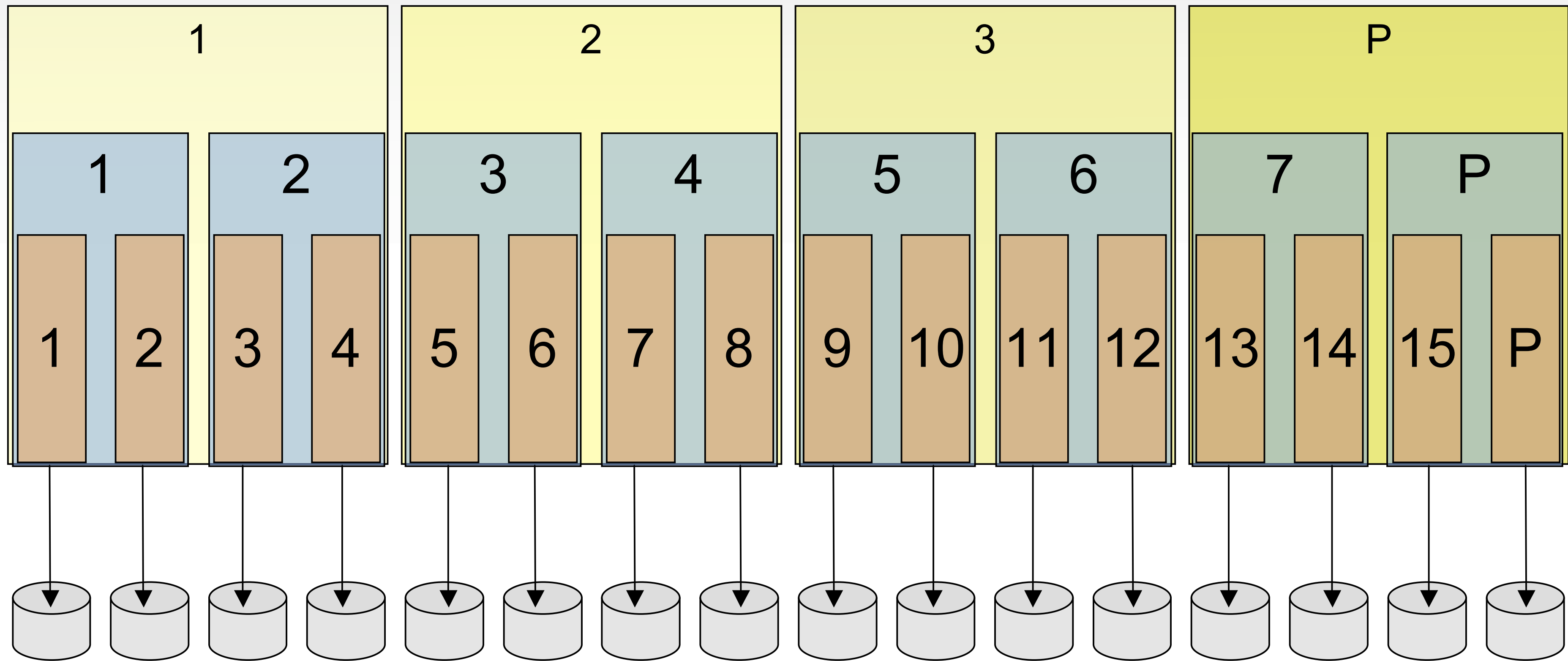
What is a Raid Vector?

"A queue from which a disk or set of disks may process incoming write requests for a given raid type without violating the recovery semantics of the raid. The raid type is determined by the client."

Given a 16-drive Zest server, a request for a:

- '3+1 raid5' would result in 4 Raid Vectors (RV) each with 4 subscribing disks.
- '7+1 raid5' -> 8 RVs @2 disks
- '15+1 raid5' -> 16 RVs @1 disk

Raid Vectors



Zest Write Techniques - RNDDP

Zest's methods for optimizing write throughput:

- Each disk is controlled by single I/O thread which heavily sequentializes incoming requests.
- **Relative, non-deterministic data placement. *(RNDDP)***
- Client generated parity.

Zest Write Techniques

Relative, non-deterministic data placement *(RNDDP)*

.. a bit more complicated

RNDDP enables highly sequential disk access by:

- Allowing for any disk in a RaidVector to process any block in that vector.
- Enabling the IO thread to place the block at the location of his choosing.

Performance is not negatively impacted by the number of clients or the degree of randomization within the incoming data streams.

Repercussions of *RNDDP*

What is the cost of doing *RNDDP*?

… Increasing entropy allows for more flexibility but more bookkeeping is required.

RNDDP destroys two inferential systems, one we care about the other is not as critical (right now).

- Block level Raid is no longer semantically relevant.
- Metadata overhead is extremely high!

Repercussions of *RNDDP*

Management of Declustered Parity Groups

...The result of "putting the data anywhere we please".

- A parityGroup handle is assigned to track the progress of a parity group (a set of related data blocks and parity block) as it propagates through the system.
- Data and parity blocks are tagged with unique identifiers that prove their association.
  - Important for determining status upon system reboot.
- Blocks are not scheduled to be 'freed' until the entire parity group has been 'synced' (Syncing will be covered shortly).

Repercussions of *RNDDP*

Declustered Parity Groups - Parity Device

A special device is used to store parity group information, generically named “parity device”.

- Parity device is a flash drive in which every block on the Zest server has a slot.
  - I/O to the parity device is highly random.
- Once the location of all parity group members is known, the parityGroup handle is written 'N' times to the parity device (where 'N' is the parity group size).
- Failed blocks may find their parity group members by accessing the failed block's parity device slot and retrieving his parityGroup handle.

Repercussions of *RNDDP*

Metadata Management

... Where did I put that offset??

Object-based parallel file systems (i.e. Lustre) use file-object maps to describe the location of a file's data.

- Map is composed of the number of stripes, the stride, and the starting stripe.
- Given this map, the location of any file offset may be computed.

RNDDP dictates that Zest has no such construct!

- Providing native read support would require the tracking of a file's offset, length pairs.

RNDDP: Interesting New Capabilities

Zest's data placement methods provide interesting feature possibilities...

Since any parity group may be written to any I/O server:

- I/O bandwidth partitioning on a per-job basis is trivial.
- Failure of a single I/O server does not create a hot-spot in the storage network.
  - Requests bound for the failed node may be evenly redistributed to the remaining nodes.

Zest Write Techniques

Zest's methods for optimizing write throughput:

- Each disk is controlled by single I/O thread which heavily sequentializes incoming requests.
- Relative, non-deterministic data placement. *(RNDDP)*
- **Client generated parity.**

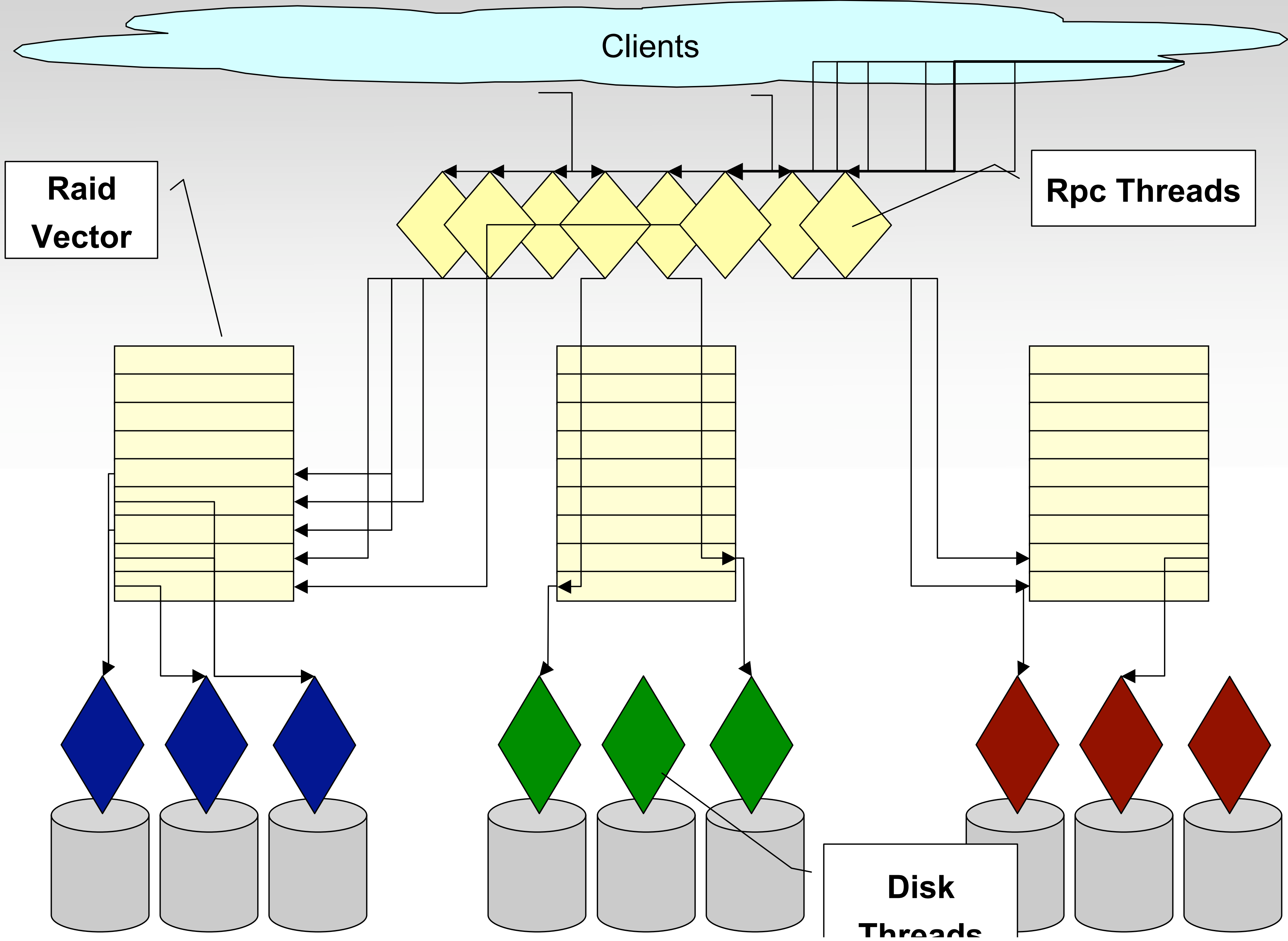
Zest Write Techniques

Client Generated Parity (and Crc)

To prevent the IO Server's raid system from being a bottleneck, redundancy information is computed at the client and handed to the server for writing.

- Data blocks are Crc'd and later verified by the Zest server during the post-processing phase.
- Data verification can be accomplished without read back of the entire parity group.

Placing these actions on the client greatly reduces the load on the Zest server at checkpoint time.


Clients
Raid Vector
Rpc Threads
Disk Threads

Post-checkpoint Processing

Checkpointing is inherently a periodic process. After a CP iteration has completed, Zest rapidly synchronizes the data with the full-featured filesystem (Lustre).

- This procedure, called '*Syncing*', is primed when completed parity groups are queued for read-back but does not actually begin until the I/O threads are finished with their write workloads.
- The 'syncer' consists of a set of threads which issue the read requests to Zest and then perform writes into Lustre.

Zest Syncer

To better understand the syncing procedure one must know how Zest correlates its block data with that of the user's file.

Specifically, we need to know:

- Which blocks are associated with a given Lustre file?
- Where block data belongs with respect to the Lustre file?

Quick jump to metadata..

Zest Metadata Management

How does Zest handle file metadata?

- Zest files are 'objects' identified by their Lustre inode number.
- On create, files are first made in the Lustre filesystem then hard-linked into an immutable section of the namespace. The inode number is used to create the path into the Zest immutable namespace.
- The identifier is returned to the client who is then responsible for presenting it to the server on a per-write basis.
- Besides the data, the client sends the list of io vectors needed to describe the write buffer.
- The identifier and the io vectors are written alongside the data buffer onto disk.

...Back to the Zest Syncer

When the syncer receives a completed read request from an I/O thread, the request contains:

- The file identifier.
- The io vectors which describe the buffer.
- The data itself.

The syncer opens the Lustre file via its immutable pathname link based on the file id and issues writes as determined by the io vectors.

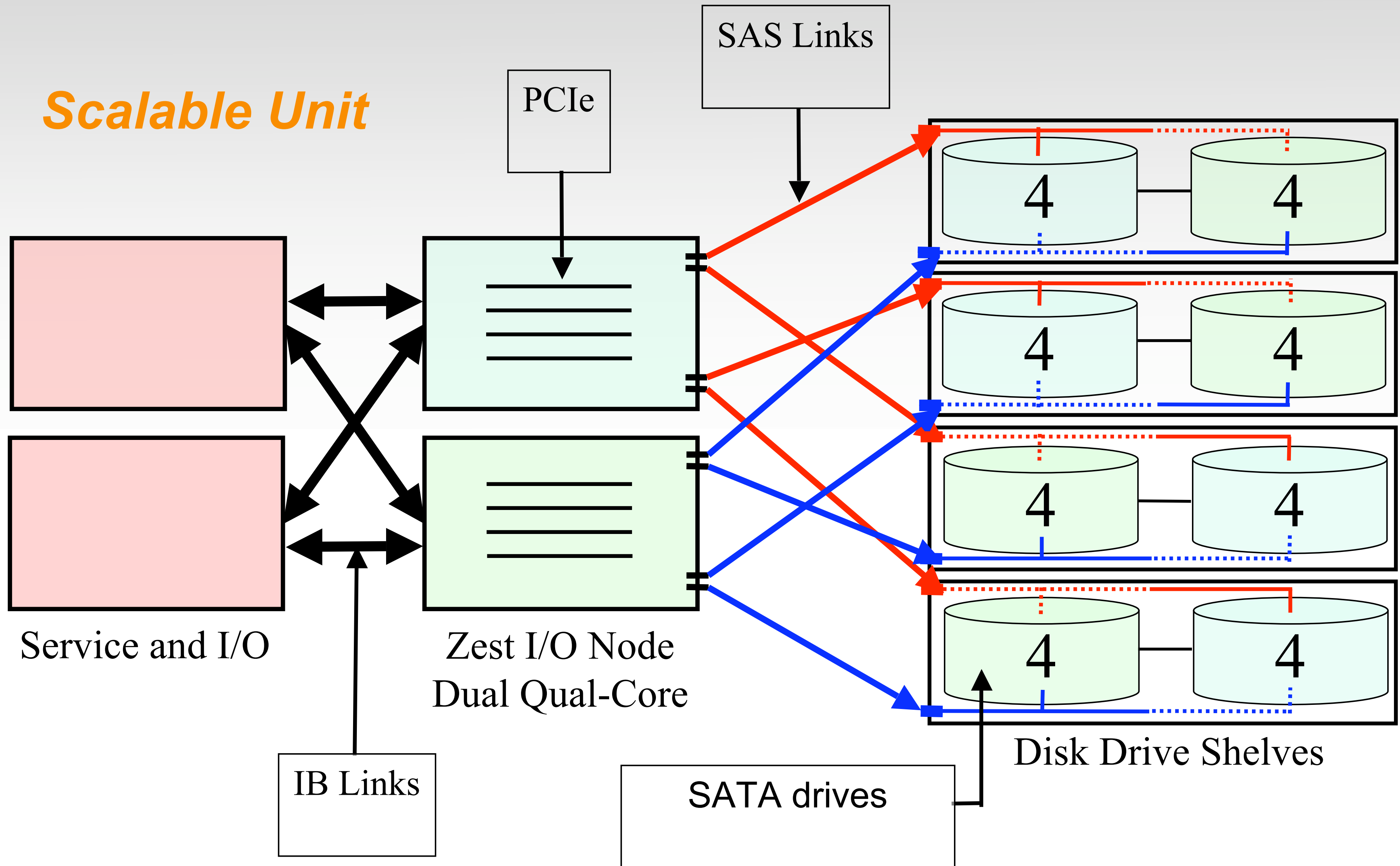
Zest Reliability

Zest provides reliability on par with a typical HPC I/O system.

- Data redundancy through Raid.
- Recoverability via multi-homed disk configuration.

Zest supports hardware configurations such as the following..

Zest Reliability



- No single point of failure

Zest Reliability Features

- Support for failover pairs.
  - Zest superblocks are tagged with UUIDs to avoid confusion in shared disk configurations.
- On reboot, corrupt or missing data is rebuilt, unsynchronized data is rectified.
- Certain modes of disk failure are easily detected and their I/O thread is quarantined.
- 'Fast rebuild' is supported.
  - When a disk fails, the Zest server has an list, in memory, of all the active blocks. Those blocks can rebuilt immediately without scanning the entire set.

Zest Networking and RPC

Zest was originally conceived to run on a XT system. Therefore it would use the service nodes as 'routers' to the external storage network.

For this reason we chose to use the Lustre Networking and RPC libraries so that Zest would be compatible with the LNET routing code (which is responsible for moving data between the compute nodes and external I/O nodes).

Zest Networking and RPC

... Bad news.

Zest runs in usermode, currently it can only utilize the *tcplnd* and the *ptllnd*.

We need the equivalent of 'UK Bridge'

. This feature will arrive soon with the introduction of user mode lustre services.

Zest Performance – Test Description

- Test consisted of sequentially writing from each PE into a separate file.
- Clients used a 5+1 Raid5 parity scheme (17% overhead)
- The number of Zest disks used was based upon the best observed incoming network bandwidth
- Two sets of results:
  - Clients on linux cluster (IB sockets-direct)
  - Clients running on an XT3 (IB ipoib)**

**No sockets-direct protocol on the XT3 SIO/routers

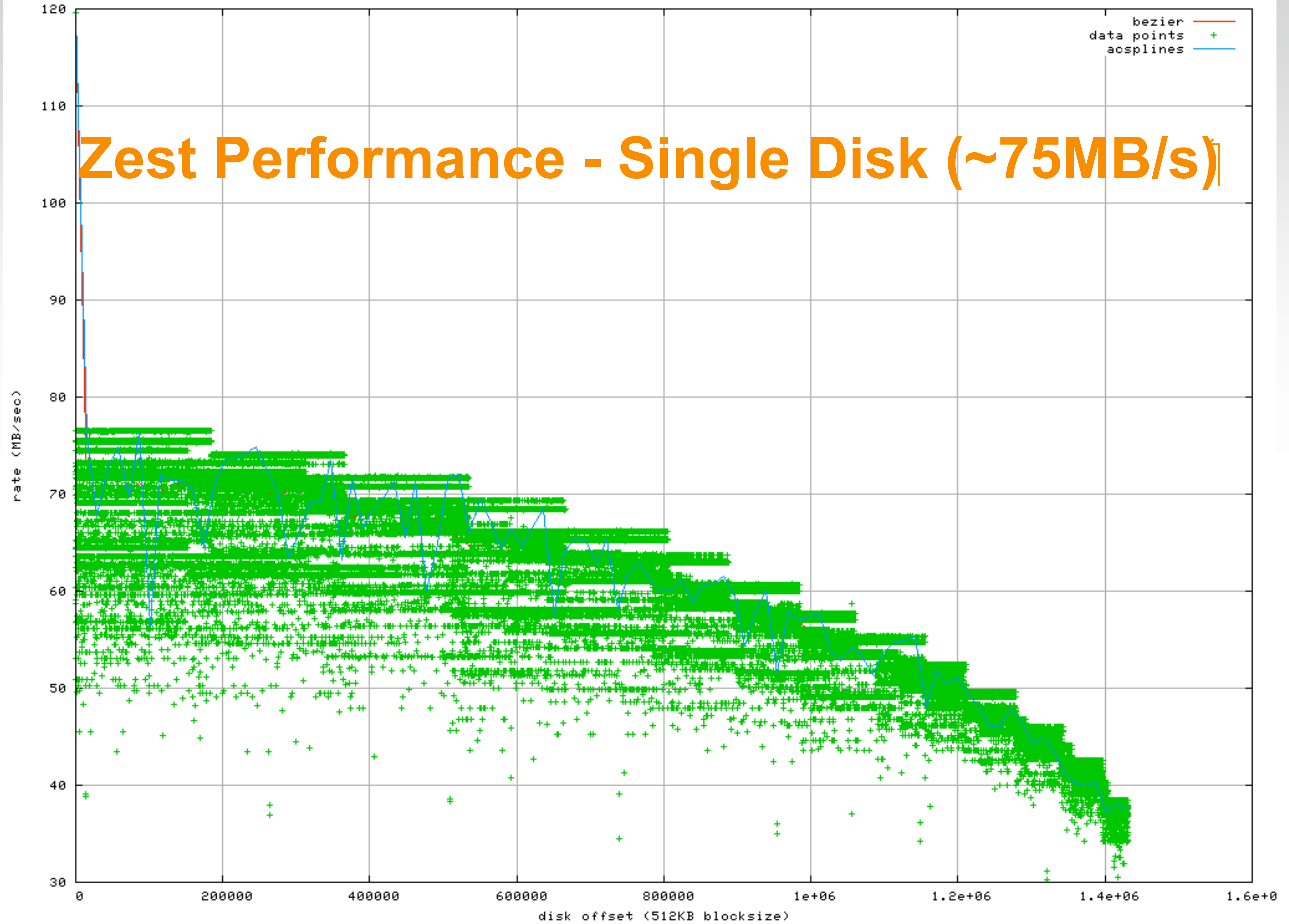
- Seems to be a problem when creating an sdp socket from kernel mode.

Zest Performance

Zest Server Hardware

- 2 x 4 Core Intel Processors
- Multiple PCI-e Busses
- 2 Sas Controllers
- 2 IB Interfaces (SDR)
- 16 Drives
  - Test uses 12 drives to match network bandwidth (~1GB/s)

Zest Performance - Single Disk (~75MB/s)

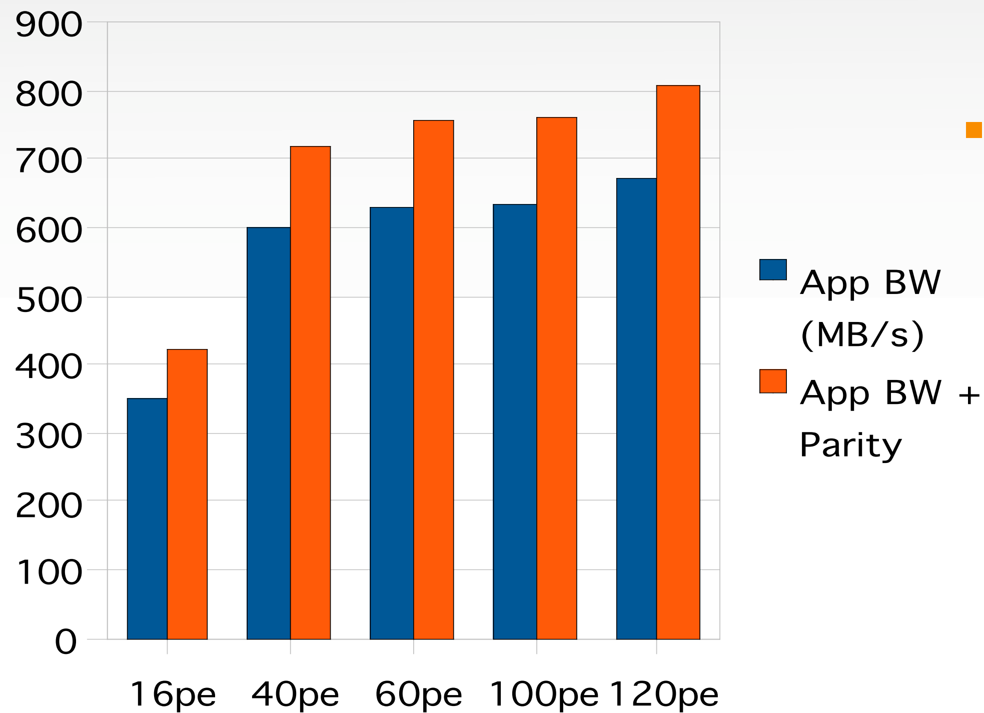


Zest Performance – Disk

By itself, the Zest backend can easily reach 90% efficiency.

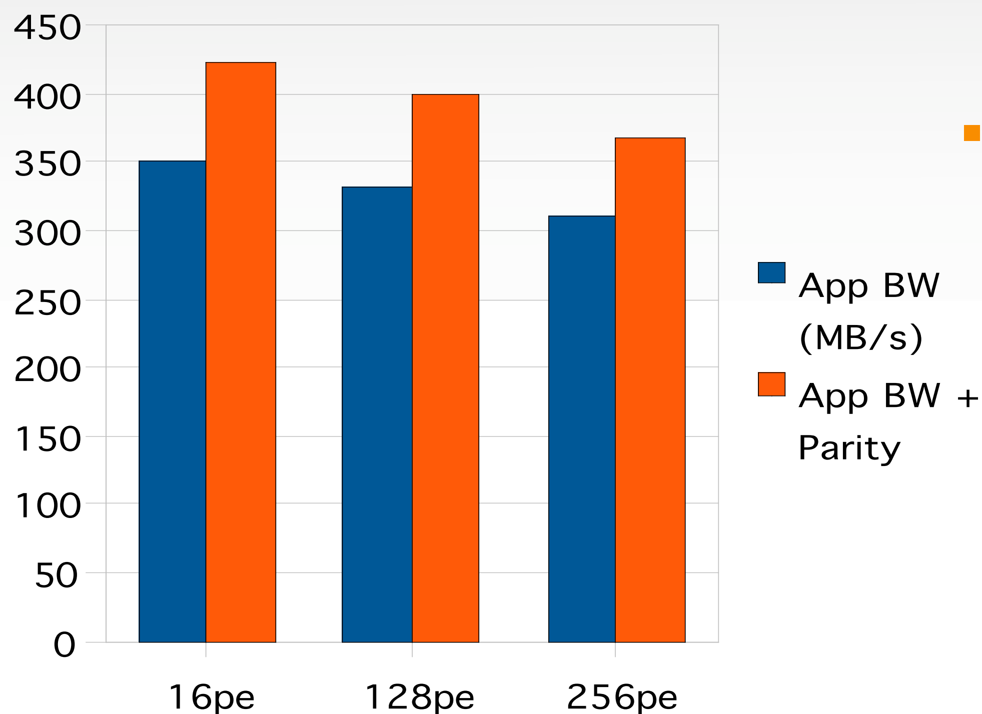
- 12 disks@840MB/s
- 16 disks@1100MB/s
- Very low CPU utilization due to zero-copy
  - About 5% of 8 cores.

Zest Performance – Linux cluster



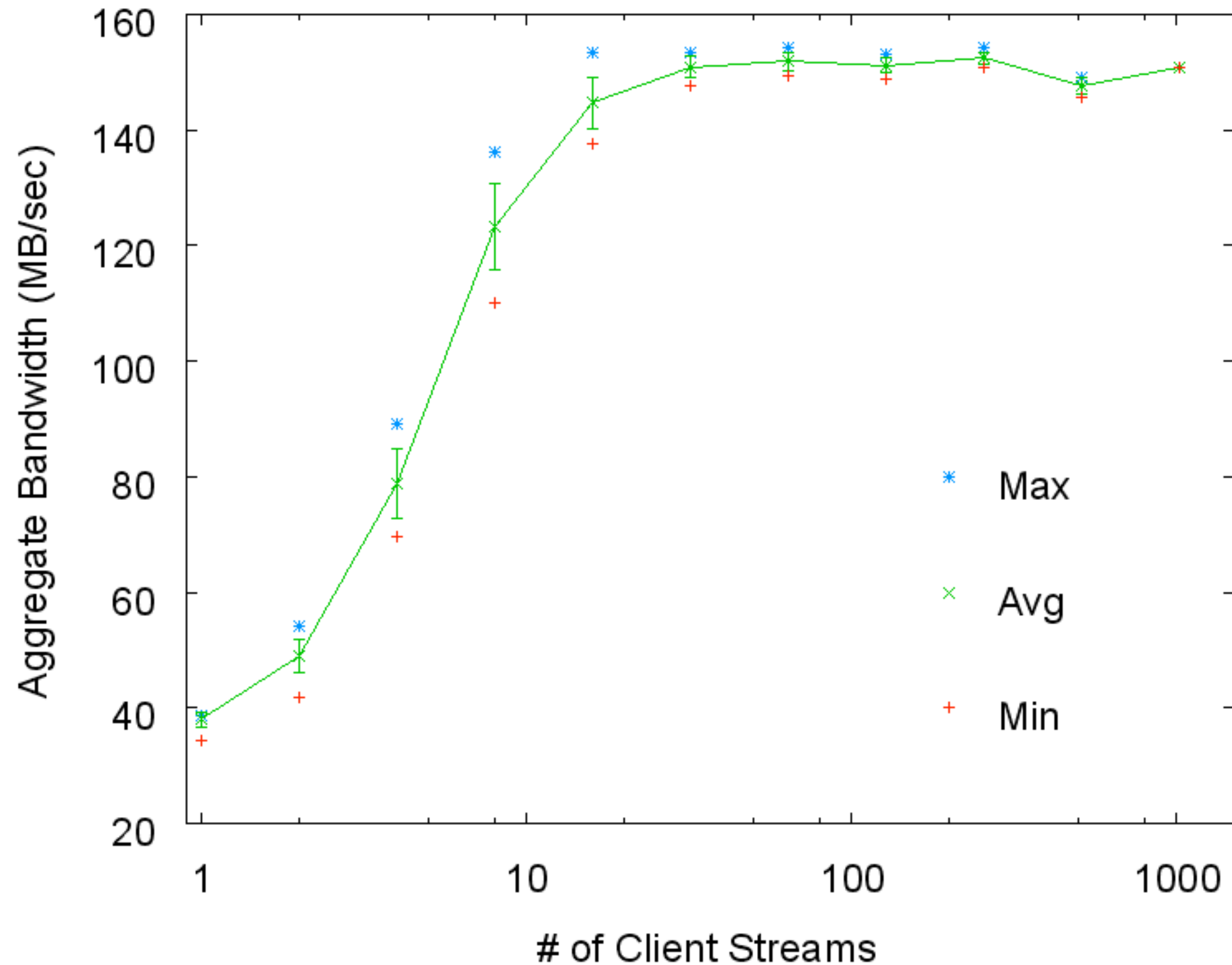
- Best case (120pe's), application saw 75% of spindle bandwidth.
- If parity overhead is ignored the transfer rate represents 89.6% of the spindle bandwidth!

Zest Performance - XT3



- Best case (16pe's), application saw 38% of spindle bandwidth.
- Ahh! The graph is going the wrong way? It's the network..
  - Ipoib is not as efficient as sockets-direct protocol.
  - Ipoib 100pe test on the linux cluster got 280MB/s.

Zest Performance - XT3



From a previous benchmark.

- Ignore the bandwidth.. (Test utilized only a single Ipoib interface.)
- Zest server shows consistent performance up to 2k clients.

Conclusions

Zest system design shows strong potential for providing low-cost, high-performance parallel I/O configurations

Network drivers need work!

- Sockets-direct protocol is not a good long term solution.
- Access to the kernel mode Lustre drivers will arrive soon.

Global Metadata Support?