A Survey of the State-of-the-art in Checkpointing

Sudharshan S. Vazhkudai

Technology Integration

National Center for Computational Sciences

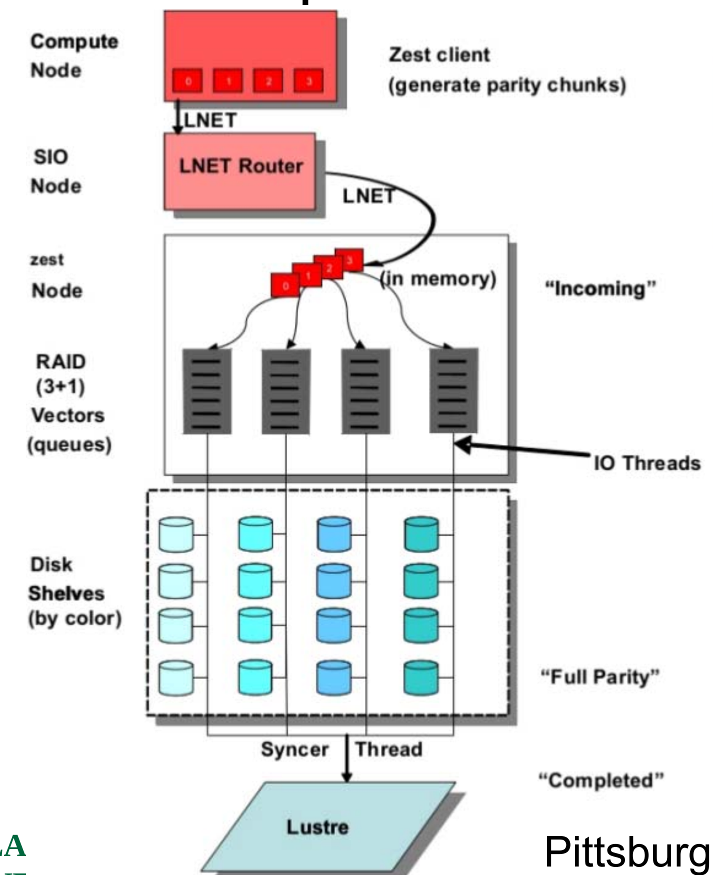
Oak Ridge National Laboratory

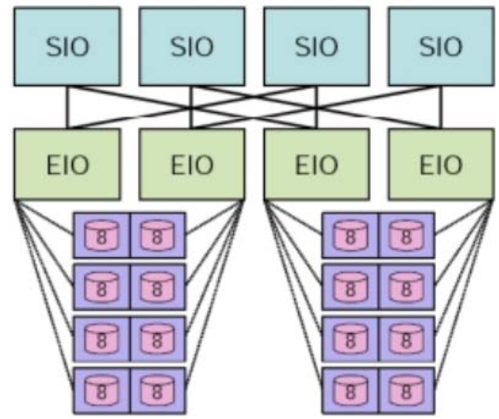
Existing Solutions for Checkpointing

- Use of intermediate resources
  – Zest, stdchk, SCR
- I/O transformation
  – PLFS, Split writing, ADIOS
- Kernel-level checkpointing
  – BLCR

Zest

- Fully utilize the potential bandwidth of disk arrays
  - Eliminate bottlenecks in PFS (parity calculation, lock, seek)
  - Absorbs I/O requests on zest nodes

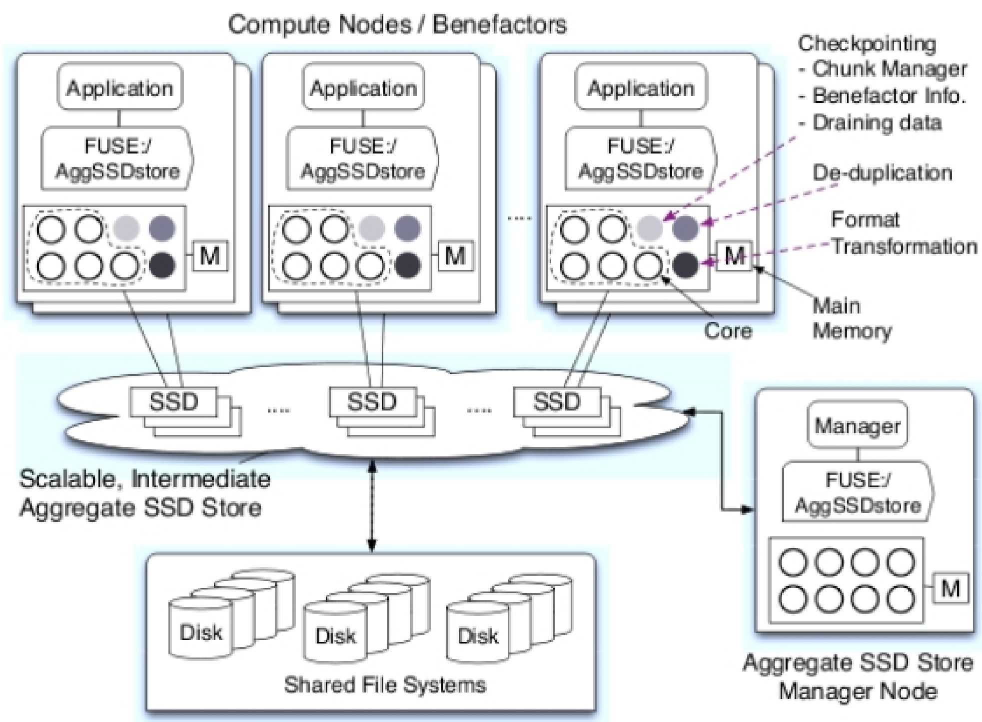


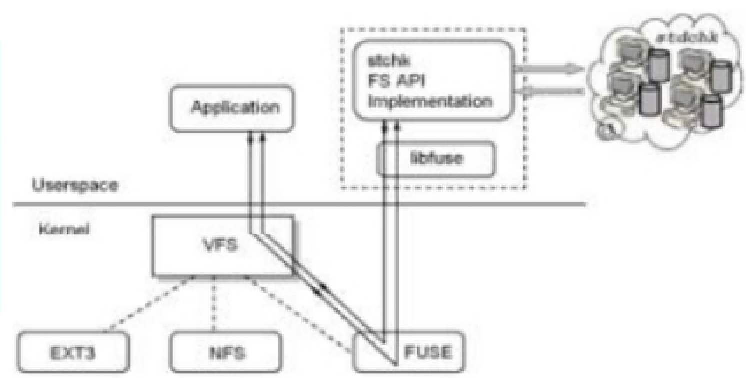


Pittsburg Supercomputing Center

stdchk

- Fast staging area exploiting idle resources
  - Dedicated file system for checkpointing
  - Providing rich features such as incremental checkpoint, replication, garbage collection, etc.





ORNL

SCR (Scalable Checkpoint/Restart)

- Advocating the use of node-local storage
  - Node-local storage is indispensable for scalable I/O
  - Utilizing both node-local storage (DRAM, flash, and disk) and PFS

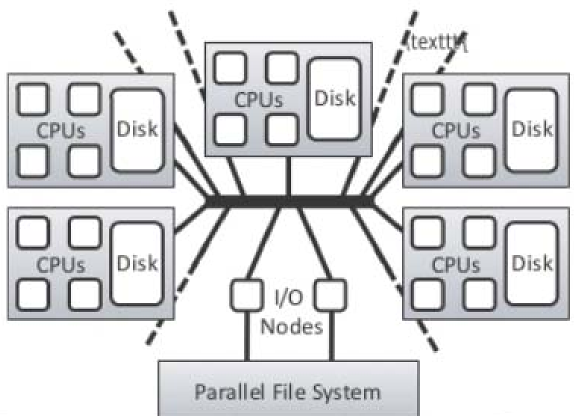


Figure 1: Traditional Cluster Design

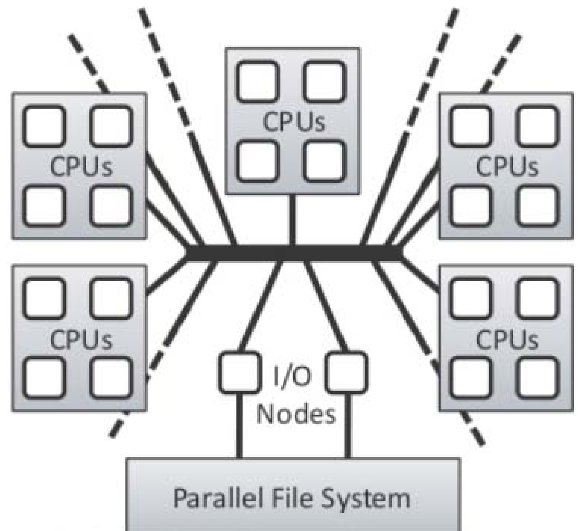
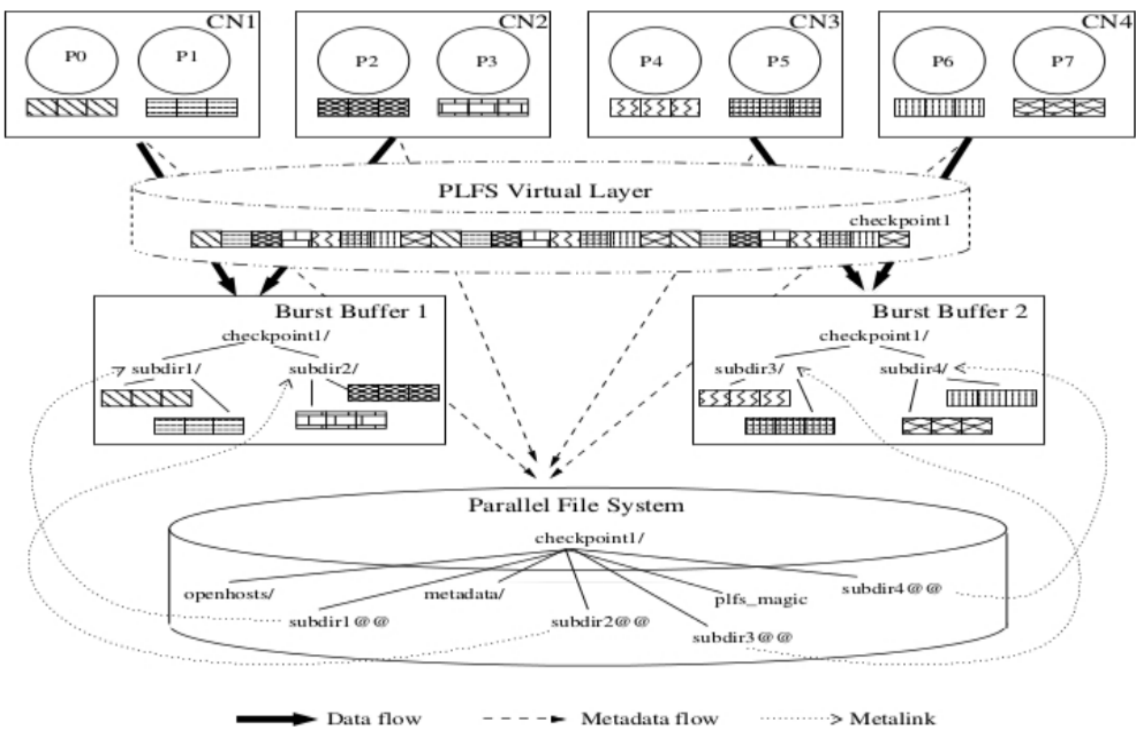


Figure 2: Modern Large Supercomputer Design

Lawrence Livermore National Lab.

PLFS

- Adapt FS-unfriendly workload (N-1 checkpoint)
  - FUSE layer transforms N-1 pattern to N-N pattern
  - Place SSDs on SIO, which absorb bursty I/O requests

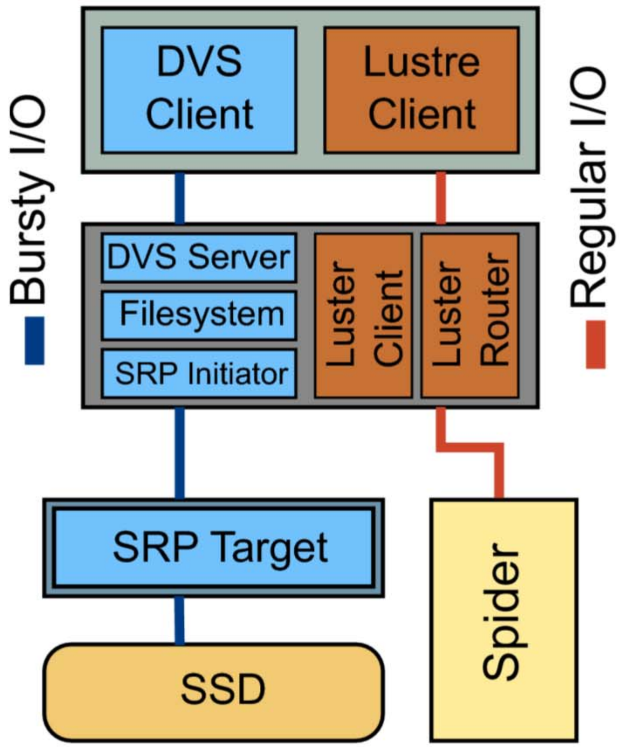


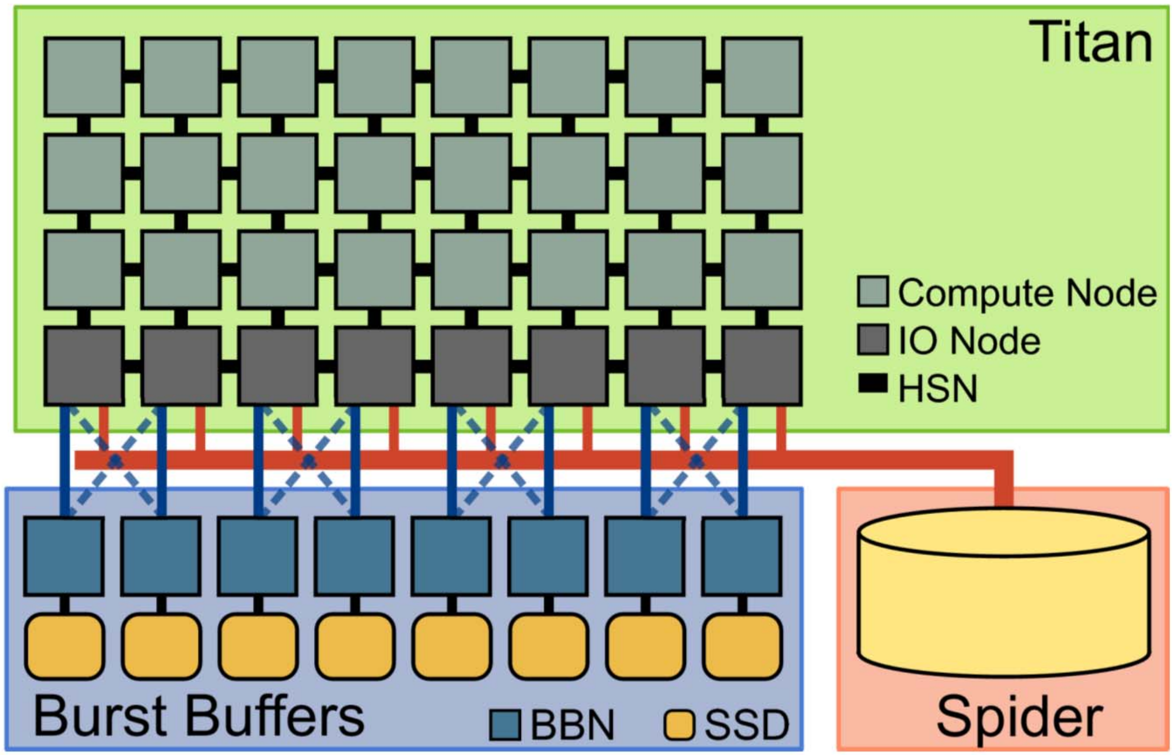
Los Alamos Notional Lab.

Berkeley Lab’s Linux Checkpoint/Restart (BLCR)

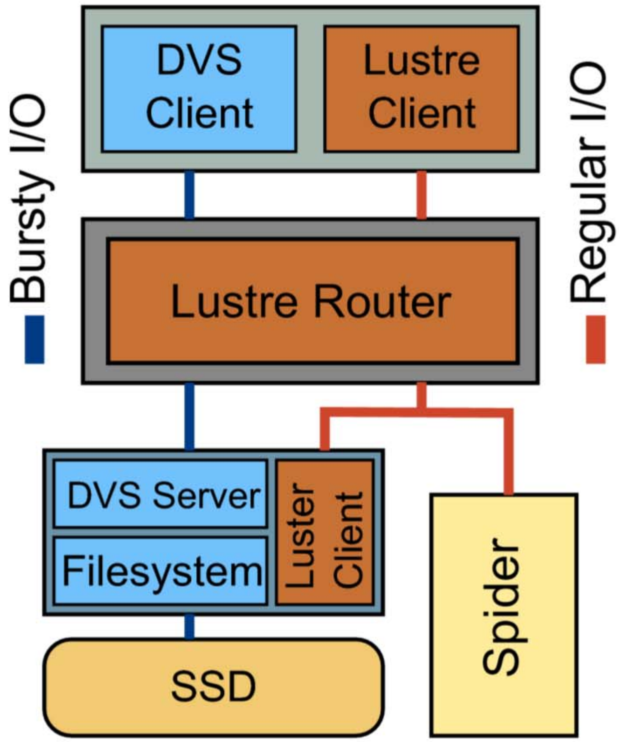
- Kernel-based C/R
  - Can save/restore almost all resources
  - Checkpoint and restart multithreaded and multiprocess applications
  - Not easily portable
- Provides interfaces to be integrated with MPI, OpenMPI or LAM/MPI
- Supports incremental checkpointing by keeping track of dirty pages

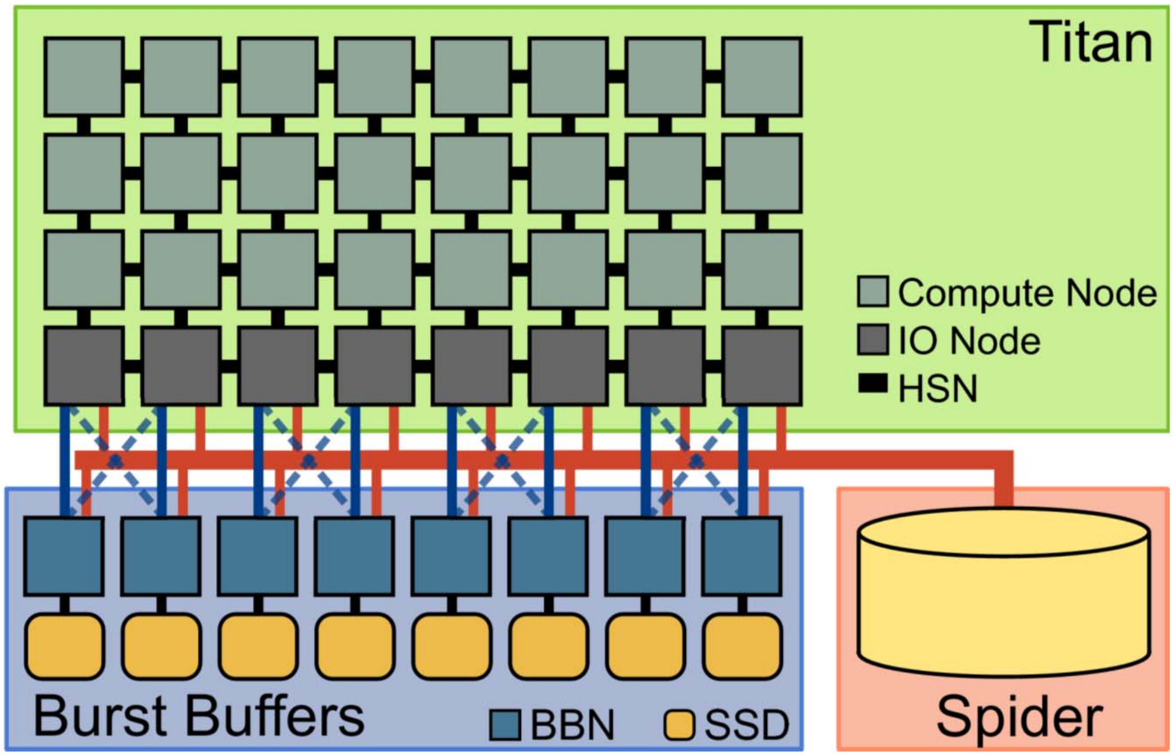
Investigating Potential Architectures for OLCF



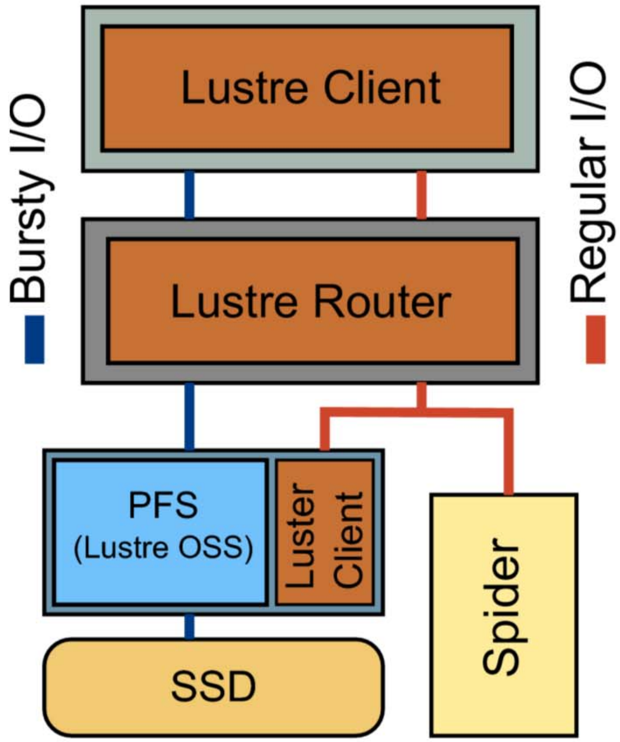


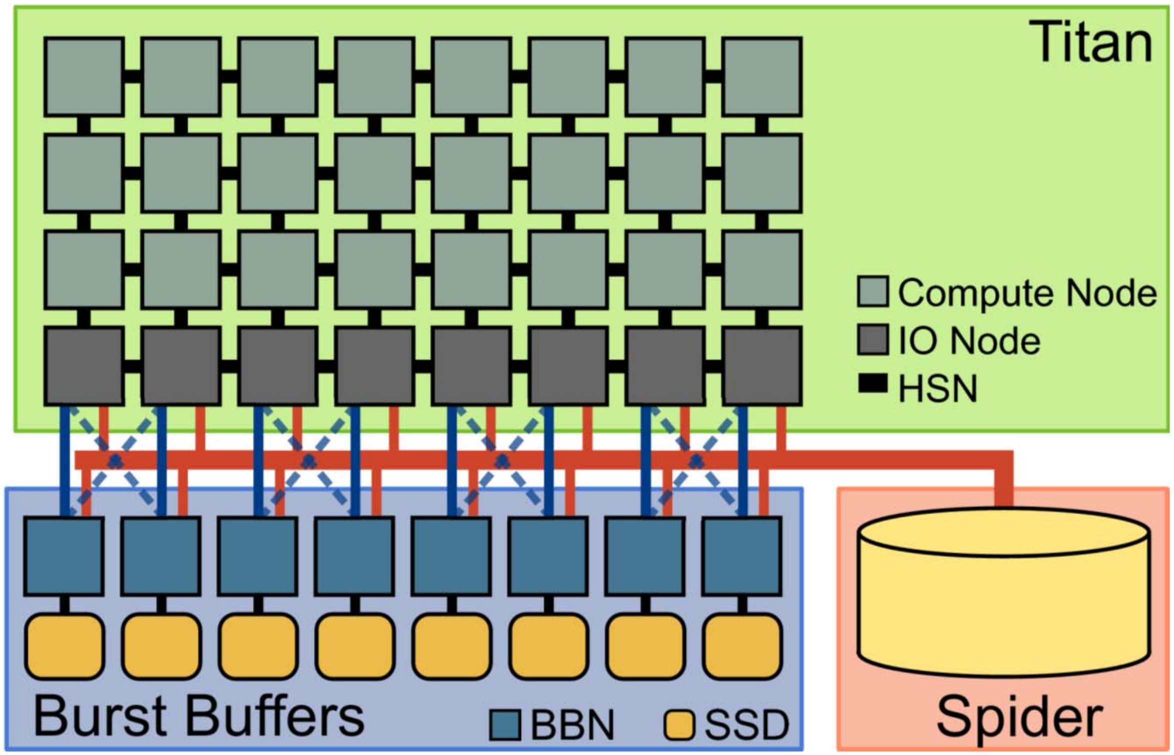
Potential Architectures for OLCF





Potential Architectures for OLCF





Burst Buffer Design Considerations

- **Capacity**: at least 3x 50% of system memory
- **Throughput**: an order of magnitude > PFS
- **Usecases**: C/R buffer, stage-in/out input/output decks, data sharing between jobs, in-situ analysis, write-through cache in the FS
- **Composition**: SSD/Flash, Disks, DRAM?
- **I/O Forwarding**: seamless I/O routing (IOFSL, DVS?)
- **Data Placement/Striping**: N-1, N-N, N-M
- **Namespace**: flat or hierarchical?
- **Draining**: When to do the draining?
- **Reliability**: level of redundancy while on the burst buffer
- **Incremental Checkpointing**: detect similarity between checkpoints

Optimal Checkpoint Frequency

- Current practice:
  - Periodic checkpoints, oblivious to machine MTBF or I/O rates
- Higher failure rate (i.e. lower MTBF) implies more frequent checkpoints
  - Potentially reduces the amount of lost work
- Longer time-to-checkpoint implies less frequent checkpoints
  - slower checkpoints increase the overall I/O overhead
  - faster checkpoints enable us to take more frequent checkpoints
- Daly, FGCS'2004 derived a theoretical optimal checkpoint period:

$$\tilde{\tau}_{\text{opt}} = \begin{cases} \sqrt{2\delta M} - \delta & \text{for } \delta < \frac{1}{2}M, \\ M & \text{for } \delta \geq \frac{1}{2}M. \end{cases}$$

M = Mean time between failures
∂ = Time to take a checkpoint

When time to checkpoint is more than half of the MTBF, you should checkpoint at every MTBF time period

Intuition: otherwise, you are *likely* to fail in the middle of every checkpoint

Failure Analysis and what it means to Checkpointing

- Temporal locality of failures
  - More failures seem to occur on the heels of a recent failure
  - Can we use this to take more frequent checkpoints immediately after a failure, and fewer checkpoints as time progresses?
- Spatial locality of failures
  - Temporal locality based guidance still based on system-wide MTBF
  - An app cares about potential failures of its node allocation
  - Is there a spatial correlation in node failures?
    - If so, can we devise a distance metric that quantifies the failure propensity of a particular neighborhood?

Application I/O Signatures and what it means to Checkpointing

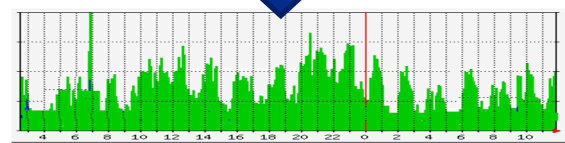
Autonomous I/O Signature Identification:

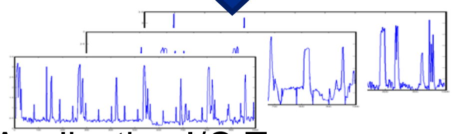
- Identify users I/O signature from server-side trace
- Zero-overhead server-side I/O usage trace data
- Scheduler's log provides info on user's runtime
- Correlate scheduler's log to trace data
- Extract common I/O features across multiple runs

Benefits:

- Identify individual user's I/O requirements
- Design and development of I/O-aware smart tools


Titan


Spider -- File System Activity

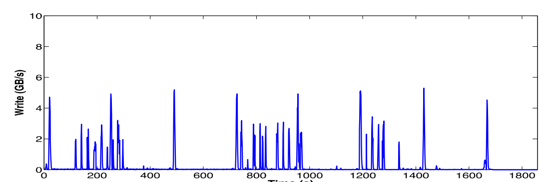

Application I/O Trace Identification

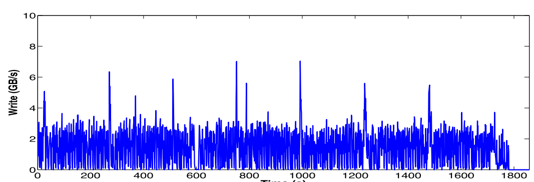
I/O-aware Smart tools

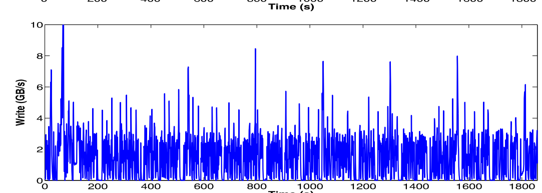
- Staggered Checkpointing
- Workload-based File System Selection
- I/O-aware Job Scheduler

Application performance variation across multiple runs

- Runtime variance
- I/O conflicts

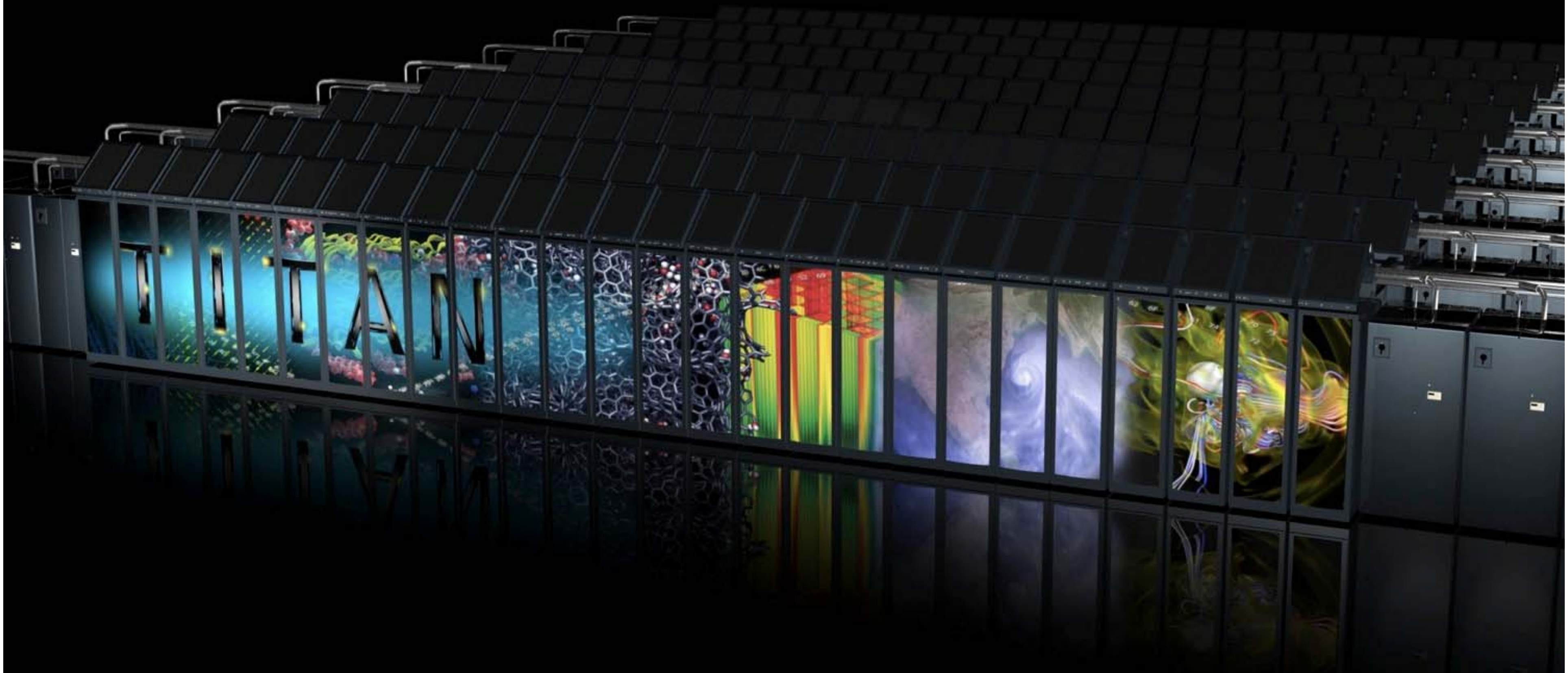






Acknowledgements

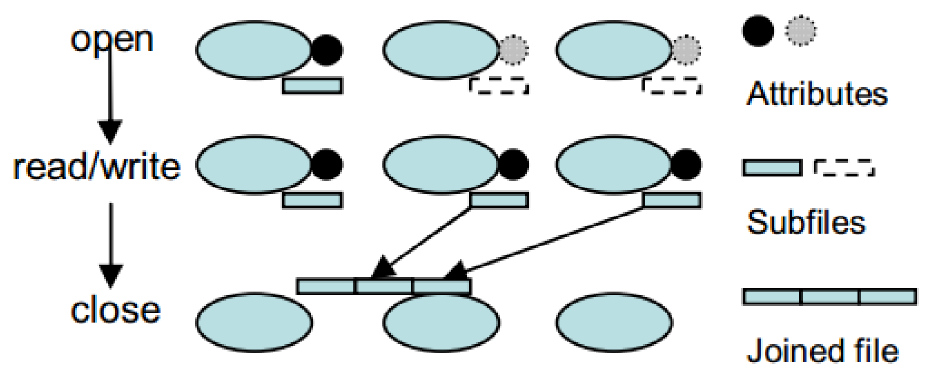
- The Technology Integration Group @ OLCF
- vazhkudaiss@ornl.gov



Backup

Sp Writ (Lustre Split Writing)

- N-1 writes suffer from MDS overhead
- Modify MPI-IO to create multiple files, which are combined on file close time



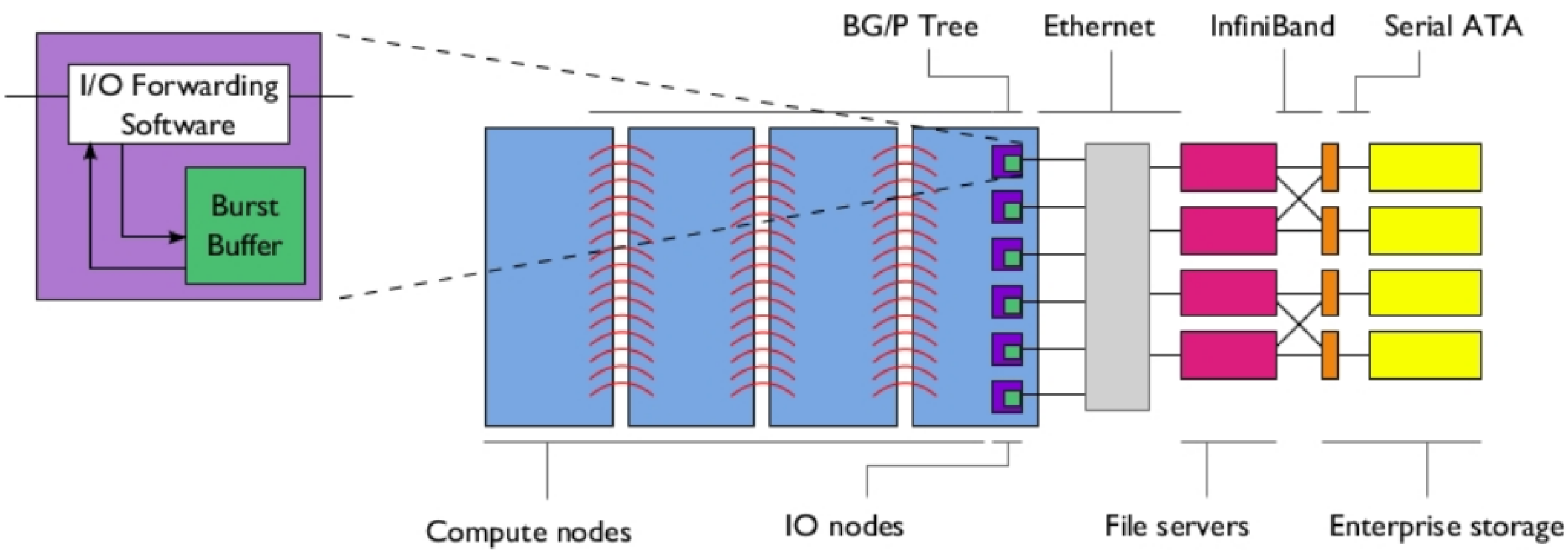
Checkpoint Storage Summary

	Interposition Technique	No Extra Resources during	No Extra Resources After	Maintaining Logical Format	Unmodified Application	Data Immediately Available
ADIOS	Library	Yes	Yes	Yes	No	Yes
stdchk	FUSE	No	No	Yes	Yes	Yes
Sp Writ	Library	Yes	Yes	Yes	No	No
ZEST	FUSE	No	No	No	No	No
PLFS	FUSE	Yes	Yes	Yes	Yes	Yes

From PLFS paper (SC'09)

Burst Buffers on Intrepid

- SSD burst buffers on SIO nodes
  - Burst buffers help reducing the scale of PFS
  - Analysis the impact via simulation



Argonne/Los Alamos National Lab., Rensselaer Polytechnic Univ.