echofs: A Scheduler-guided Temporary Filesystem to leverage Node-local NVMs

Alberto Miranda, Ramon Nou
Barcelona Supercomputing Center (BSC)
alberto.miranda@bsc.es, ramon.nou@bsc.es

Toni Cortes
Barcelona Supercomputing Center (BSC)
Universitat Politècnica de Catalunya
toni.cortes@bsc.es

***Abstract*—The growth in data-intensive scientific applications poses strong demands on the HPC storage subsystem, as data needs to be copied from compute nodes to I/O nodes and vice versa for jobs to run. The emerging trend of adding denser, NVM-based burst buffers to compute nodes, however, offers the possibility of using these resources to build temporary filesystems with specific I/O optimizations for a batch job. In this work, we present echofs, a temporary filesystem that coordinates with the job scheduler to preload a job's input files into node-local burst buffers. We present the results measured with NVM emulation, and different FS backends with DAX/FUSE on a local node, to show the benefits of our proposal and such coordination.**

I. Introduction

Though current high-performance computer (HPC) systems perform on the order of tens of hundreds of petaFLOPs, the recent growth in scientific modeling, simulation and analytic workloads poses strong demands on the I/O subsystem: even if current systems can process the data quickly, HPC clusters still need to load input data into the system and write intermediate checkpoints and generated results out of it. I/O performance, however, has struggled to keep up with computing performance due to interfering concurrent applications, suboptimal access patterns, and missing application interfaces [1].

To mitigate this, *burst buffers* have begun to be integrated into the HPC storage architecture. A burst buffer is a piece of fast storage hardware that is made available to a compute node, either locally (node-local burst buffers) or remotely through fast interconnection fabrics (remote/shared burst buffers). By writing data to burst buffers, applications perceive that data has been quickly saved to persistent storage, even if this storage is not its ultimate destination. Also, upcoming non-volatile memory (NVM) technologies such as Intel® and Micron®'s 3DXPoint™ provide significantly improved performance over traditional storage media with the added benefit of data persistence. As such, there is a growing interest in combining fast conventional DRAM with the slower but denser NVM [2], [3], [4], leveraging these hybrid memory layouts to improve I/O. Nonetheless, the inclusion of burst buffers in the storage architecture has not yet been well-studied nor have any standard APIs been defined, which means that researchers often need to spend time trying to find the best way to use them in their applications.

This usage of burst buffers also has the issue that data dependencies between jobs are not conveyed to the storage stack: if Job A generates output data that is going to be reused by Job B, there is no way for users to influence the job scheduling process so that this data can be kept in the burst buffers until Job B starts. Worse yet, given that in many HPC systems data storage is separate from the main system,that the I/O stack remains essentially a black box for today's job schedulers,Job A's output data could be staged out to the cluster's centralized storage and, at some point in the near future, staged back into a new set of compute nodes for Job B, which might end up including some of the nodes allocated to Job A. Thus, job schedulers could favor a better utilization of HPC resources by having information about these workflows.

Moreover, new optimization opportunities appear if applications are able to convey their intended usage of data: for instance, temporary files could be kept in the compute nodes' burst buffers until the job completes, at which point they could be safely deleted without having to stage them out to centralized storage. Also, accesses to centralized storage could be arranged to benefit overall I/O. Though this case-by-case optimizations are not applicable to the general purpose filesystems conventionally used in HPC systems such as GPFS [5], they can be implemented in a short-lived filesystem linked to the HPC job lifetime, that relies on the faster burst buffers for storage and coordinates with the job scheduler and the application to avoid common performance pitfalls.

In this work we present a user-level filesystem called `echofs` that resides between applications and the cluster's centralized storage and leverages the local burst buffer hardware available to compute nodes. The filesystem relies on the job scheduler to convey any data dependencies of the upcoming batch job, and then uses this information to preload/offload these dependencies to/from each local node. Also, `echofs` offers a POSIX interface and internally manages the accesses to the different burst buffers available to a compute node (e.g. SSDs or NVM), so that researchers and legacy applications can transparently benefit from the technological advantages of new hardware without dealing with its details.

II. Architecture and Implementation

The Echo Filesystem (`echofs`) is a temporary filesystem layer that resides on compute nodes between applications and the centralized parallel filesystem (PFS), and that lives for as long as a batch job (or a series of related batch jobs) needs it. This additional storage layer is added to the compute nodes I/O stack with several objectives in mind: (1) to hide

the complexity of the different levels of storage that may be available to an application running in a compute node (e.g. NVM, SSDs, or HDDs) by aggregating them into a virtual storage device with a single mount point; (2) to reduce the contention of the PFS by serving application I/O from local storage resources; (3) to provide a POSIX interface so that applications can interact with (and benefit from) new storage technologies with the usual APIs, without having to write new code; (4) to coordinate with the job scheduler to capture the data dependencies of submitted jobs, laod them into node-local burst buffers and improve application-perceived I/O.

Thus, when a batch job is submitted, the job scheduler determines the initial set of data dependencies,[1] allocates a set of compute nodes on the HPC cluster, constructs instances of `echofs` across these nodes and communicates the captured information to the filesystem using a specialized API. At this point, the filesystem's components running on the compute nodes coordinate to prefetch all input data from the PFS. Once the preload process completes, the batch job can be started and the application can use the standard POSIX I/O functions to access the data contained in the filesystem. Supporting the POSIX I/O API allows legacy applications that cannot be modified to benefit from newer storage technologies, without having to deal with new interfaces or semantics. The POSIX interface itself is implemented using the FUSE framework[2], so that legacy applications can run directly on `echofs`.

Any newly created files are written directly to the burst buffers and are offloaded back to the PFS in a location predefined by the user when the job completes. Note, however, that this is done *only if* the user or the job scheduler explicitly requested (tagged) these files to be *persistent* in the configuration file. Moreover, files can also be tagged as *reusable*, which effectively keeps them in the burst buffers to allow upcoming jobs to reuse the generated data. This helps reduce some of the I/O contention on the PFS, by eliminating the need to persist any temporary files and/or checkpoints generated by an application, at the expense of shifting the responsibility of "tagging" output files to the scheduler/user. These *usage hints* can be statically conveyed to `echofs` by either the job scheduler or the application through a specialized API that allows the preloading, offloading and tagging of path.

a) Metadata management: The management of POSIX metadata operations in a PFS (such as `open()` or `stat()`) is challenging and often relies on several metadata servers in charge of keeping metadata consistent, providing file handles, and giving/revoking rights to clients to update and cache file information. Thus, since metadata operations are in the critical path of many I/O operations, metadata servers often become a serious bottleneck when concurrent requests from large numbers of clients [6], [7] are received. Since `echofs` focuses on serving I/Os to a single parallel application, which "temporarily owns" a set of storage resources exclusively dedicated to it, it has advantages w.r.t. general purpose PFSs: (1) since its lifetime is limited, `echofs` needs to support only application processes running in the context of the corresponding batch job, which, when compared to the large number of clients that a general purpose PFS needs to serve, reduces scalability requirements considerably; (2) since the filesystem runs with the uid and gid of its owner, and no other users are allowed to access the mount point, permission validation only needs to be enforced when importing input files into the `echofs` instance, and thus is effectively removed from the I/O critical path.

Thus, when mounted, `echofs` creates a typical POSIX directory-based namespace upon which it imports any input data specified in the batch job script, at the locations defined by the user. Given that the namespace does not need to outlive the application, `echofs` stores metadata into a specialized in-memory hash map in order to serve operations as fast as possible. Also, since permissions no longer need to be checked for each operation, validation is removed from the path resolution process, further reducing the latency of metadata operations.

b) Data management: In the prototype, NVM accesses are managed using Intel's PMDK Library [3] to create memory-mapped files on top of a filesystem with DAX capabilities, which allows for direct access to the hardware with byte granularity. Using this library has the advantage that the same interface can be used for NVM, SSDs and HDDs, as long as the underlying filesystem supports `mmap()`. Since the PMDK library is intended to work with memory-mapped files, `echofs` relies on *NVM segments* that represent a contiguous range of file data that is backed by a PMDK-mapped file. Thus, when a file is preloaded into `echofs`, a new segment is created and populated with the contents of its PFS counterpart. The segment is then registered into a per-file *segment tree* which allows resolving offsets in $\mathcal{O}(\log n)$. When files need to grow (or are created anew), new segments of a pre-configured size are allocated and appended to the tree. This allows files to grow as needed and also allows easily supporting sparse files to reduce storage overhead. When the filesystem is unmounted, the segments for persistent files are moved and consolidated to the PFS. Segments for reusable files are consolidated into a unified file so that they can be used by future processes. Persistence of write operations is internally managed by `echofs` by using the appropriate `msync()` and `drain()` calls through PMDK.

c) SLURM integration: The `echofs` prototype relies on a modified SLURM [8] implementation that allows users to provide *static data usage hints* in their job submission commands, so that it can later pass them to `echofs` when it is mounted through its companion API. Moreover, applications can also use the API if they are willing to provide *dynamic data usage hints* while running, though it's not mandatory.

III. Evaluation

All tests were run on a cluster with 10 nodes interconnected by a 1GbE link. Each node had a 4-core Intel® Xeon® E5-2609v2 2.50GHz processor, 32GiB DRAM, and a 400GB

[1] That is, the files that are going to be used by the application, the information of which will have been provided by the user upon submission.

[2] https://github.com/libfuse/libfuse

[3] https://pmem.io

Intel P3500 SSD. A Lustre 2.8 filesystem was deployed using 1 node for the Metadata Service (MDS), and 2 nodes as Object Storage Targets (OSTs), with the aforementioned SSDs as backend. The NVM hardware itself is emulated using a DRAM backend, on top of which an ext4 filesystem with DAX support is mounted. Some tests were also run with a 375 GB Intel Optane™ SSD to get an insight on actual NVM hardware performance. To measure performance, we use the well-established IOR and MDTEST microbenchmarks.[4]

a) Metadata performance: We generate a metadata workload for `echofs` using MDTEST and compare its performance against the metadata subsystems of ext4 (both with a local HDD and a local SSD) and Lustre. We instruct MDTEST to create 2.000.000 files/directories in a single parent directory, and run it with 1 and 4 processes. The mean results, with 10 repetitions, can be found in Table I. We observe that, as expected, `echofs` shows overall better performance than its counterparts, since it has been designed precisely to avoid the metadata issues of shared filesystems such as Lustre, and the latency of RAM-backed metadata should be better than HDD/SSD-backed metadata. The only exception can be observed with `stat()` operations. In this case, performance is limited by the maximum number of operations per second supported by FUSE. Though `stat()` scales nearly linearly on HDDs, SSDs and Lustre, `echofs` needs 2x more context switches than a kernel filesystem, due to FUSE's architecture.

b) Data performance: We exercise `echofs` using the IOR microbenchmark with different storage backends and with two different FUSE versions: 2.9 and 3.0, patching the kernel to increase the internal buffer size to 12MiB. We instruct IOR to create a file/set of files of 1GiB inside a directory, respectively performing shared/non-shared operations upon them using a 128KiB transfer size and 4 processes. Table II summarizes the results for different storage devices compared against the best-performant `echofs` results. As expected, `echofs` obtains more performance than using traditional storage, as it uses a memory-like technology, but is also more performant than local SSDs. In the case of Optane, we observe how it surpasses SSD's speed but falls short w.r.t. performance using memory.

We found discrepancies between the `echofs` results when using different DAX-enabled filesystems as data backends. On Figure 1, we show an ECDF (Empirical Cumulative Distribution Function) of the bandwidth obtained with different DAX filesystems (ramfs, ext3, ext4 and XFS) and FUSE versions. XFS and ramfs showed good throughput, but we observed low performance with ext4, the *de facto* filesystem used to test NVM libraries. Note that the ECDF lines are mostly vertical, which means that bandwidth is fairly stable in all runs.

c) Coordinated echofs stage-in/out vs. direct IOR in PFS: To prove that using `echofs` can reduce the overall PFS I/O contention, we ran a proof-of-concept experiment where 3 16GiB files were created in Lustre, which were then sequentially read and rewritten, concurrently, from 3 different nodes (see Table III). Relying on the PFS, so that data always stays in-sync as is typically done in HPC clusters, required 835.5 seconds to finish the experiment. Relying on `echofs`' capabilities to stage-in and stage-out automatically took ≈715.3 seconds with concurrent stage-in phases (a 14.38% improvement), and 583.2 seconds with a simple SLURM heuristic to interleave stage-in phases (a 30.19% improvement). Benefits were more significant if data was reused: using 2 nodes and 2 2.5GiB files, required ≈240.5 seconds to execute 10 iterations directly on the PFS, and 71.8 seconds in `echofs` a 70.14% improvement.

[4]https://github.com/hpc/ior

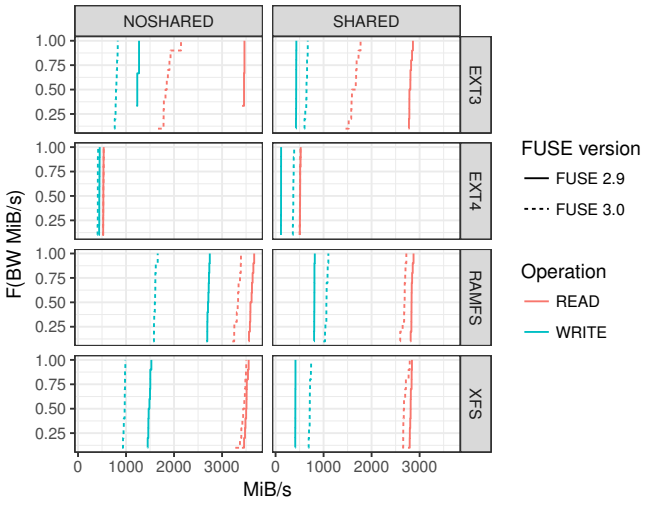

Fig. 1. Throughput using different FUSE versions and DAX filesystems

IV. Related Work

Using burst buffers to improve the I/O performance of HPC applications was proposed in order to cope with the forecasted explosion of data-centric computing in future exascale supercomputers [9]. Hence, several works have been published exploring remote, shared burst buffers: the SCR group is currently trying to strengthen the support for SCR by developing a multi-level checkpointing scheme on top of burst buffers [10]. DataDirect Networks is developing the Infinite Memory Engine (IME) as a burst buffer layer that provides real-time I/O for scientific and research applications [11]. BurstMem [12] extends and modifies Memcached to work on burst buffers and provide a mechanism of coordinated data shuffling and flushing to the PFS, while IBIO [13] explores how burst buffers can be used to improve resiliency by testing a wide range of checkpoint/restart strategies. Conversely, TRIO [14] proposes an orchestration framework to efficiently move large checkpointing datasets to the PFS with efficiently utilized storage bandwidth and reduced job I/O time.

With regards to node-local burst buffers, PLFS offers support by relying on a global filesystem (e.g. Lustre) to manage metalinks. It is able to deliver fast, scalable write performance and is also able to achieve scalable read performance by using the collective services of its delegators [15]. Similarly to our proposal, BurstFS [16] proposes an ephemeral burst buffer-based filesystem with the same lifetime as an HPC job. BurstFS organizes the metadata on any data written to local burst buffers into a distributed key-value store, and localizes

TABLE I
MDTEST RESULTS IN OPERATIONS PER SECOND

	Create Dir	Stat Dir	Remove Dir	Create File	Stat File	Remove File
HDD (1P)	1324	515 096	1913	1810	370 508	1810
HDD (4P)	1460	1 936 912	1716	2067	1 412 165	1751
SSD (1P)	10 744	524 500	23 534	19 572	368 323	25 944
SSD (4P)	11 409	1 821 136	22 514	21 508	1 382 580	29 174
Lustre (1P)	7184	463 454	15 712	9396	457 911	12 990
Lustre (4P)	9541	1 824 400	11 643	9894	1 819 549	10 651
echofs (1P)	30 675	37 746	34 445	20 258	35 778	30 658
echofs (4P)	27 731	84 315	32 294	22 047	81 646	30 257

TABLE II
IOR RESULTS (HDD, SSD, OPTANE AND BEST PERFORMANT ECHOFS OPTION)

	SHARED		NO SHARED	
	Read (MiB/s)	**Write (MiB/s)**	**Read (MiB/s)**	**Write (MiB/s)**
Lustre (1GbE)	111.39	112.81	145.39	113.18
HDD	114.93	140.57	89.77	123.78
SSD	235.12	288.97	232.39	291.96
Optane	1035.39	616.43	752.72	733.74
FUSE 2.9 – ramfs	2835.24	807.37	3606.85	2710.27

TABLE III
STAGE-IN VS DIRECT PFS I/O RESULTS

	Time to completion	Improvement
PFS I/O 3x16GiB	835.5 s	—
Uncoordinated 3x16GiB	715.3 s	14.38 %
Interleaved 3x16GiB	583.2 s	30.19 %
PFS I/O 2x2.5GiBx10 its	240.5 s	—
echofs+reuse I/O 2x2.5GiBx10 its	71.8 s	70.14 %

reads and writes using co-located I/O delegation and server-side read clustering. The main differences between `echofs` and BurstFS is that `echofs` coordinates with the job scheduler and that it offers an API for applications to convey usage hints.

V. CONCLUSIONS

In this paper, we propose a temporary filesystem called `echofs` that leverages local burst buffers and an integration with the batch job scheduler to preload data across all nodes that participate in a parallel computation. The filesystem lives as long as the batch job that spawned it requires, and hides the complexity of the burst buffer management from applications, which can access the data space using a POSIX interface.

Performance results show that, despite the limitations imposed by the current version of the FUSE library, combining the information provided by the job scheduler with the performance advantage of NVM-based burst buffers allows for interesting I/O performance gains and, more importantly, for better-behaved I/O access patterns from the point of view of the PFS, which leads to reduce overall I/O contention. Thus, since filesystems are typically not informed about the intent of parallel applications, we believe that there is value in improving the communication channels between the HPC infrastructure, the parallel applications, and the temporary filesystem, which would allow for better I/O optimizations.

ACKNOWLEDGMENT

This work was partially supported by the Spanish Ministry of Science and Innovation under the TIN2015–65316 grant, the Generalitat de Catalunya under contract 2014–SGR–1051, as well as the European Union's Horizon 2020 Research and Innovation Programme, under Grant Agreement no. 671951 (NEXTGenIO). Source code available at https://github.com/bsc-ssrg/echofs.

REFERENCES

[1] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, p. 8, 2011.

[2] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2010, pp. 385–395.

[3] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 73–80.

[4] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 707–722.

[5] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, 2002, pp. 231–244.

[6] D. S. Roselli, J. R. Lorch, T. E. Anderson *et al.*, "A comparison of file system workloads." in *USENIX annual technical conference, general track*, 2000, pp. 41–54.

[7] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. Long, and T. T. McLarty, "File system workload analysis for large scale scientific computing applications," in *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2004, pp. 139–152.

[8] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[9] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2012, pp. 1–11.

[10] SCR, "SCR: Scalable Checkpoint/Restart for MPI." [Online]. Available: https://computation.llnl.gov/project/scr

[11] DDN, "Infinite Memory Engine." [Online]. Available: https://www.ddn.com/products/ime-flash-native-data-cache/

[12] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, "BurstMem: A high-performance burst buffer system for scientific applications," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 71–79.

[13] K. Sato, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, N. Maruyama, and S. Matsuoka, "A user-level infiniband-based file system and checkpoint strategy for burst buffers," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 2014, pp. 21–30.

[14] T. Wang, S. Oral, M. Pritchard, B. Wang, and W. Yu, "TRIO: burst buffer based I/O orchestration," in *2015 IEEE International Conference on Cluster Computing*. IEEE, 2015, pp. 194–203.

[15] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring, "Jitter-free co-processing on a prototype exascale storage stack," in *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2012, pp. 1–5.

[16] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 69.