Software-Defined Storage for Fast Trajectory Queries using a DeltaFS Indexed Massive Directory

Qing Zheng[†], George Amvrosiadis[†], Saurabh Kadekodi[†], Garth A. Gibson[†], Charles D. Cranor[†], Bradley W. Settlemyer[‡], Gary Grider[‡], Fan Guo[‡]
[†]Carnegie Mellon University, [‡]Los Alamos National Laboratory

ABSTRACT

In this paper we introduce the Indexed Massive Directory, a new technique for indexing data within DeltaFS. With its design as a scalable, server-less file system for HPC platforms, DeltaFS scales file system metadata performance with application scale. The Indexed Massive Directory is a novel extension to the DeltaFS data plane, enabling in-situ indexing of massive amounts of data written to a single directory simultaneously, and in an arbitrarily large number of files. We achieve this through a memory-efficient indexing mechanism for reordering and indexing data, and a log-structured storage layout to pack small writes into large log objects, all while ensuring compute node resources are used frugally. We demonstrate the efficiency of this indexing mechanism through VPIC, a widely-used simulation code that scales to trillions of particles. With DeltaFS, we modify VPIC to create a file for each particle to receive writes of that particle's output data. Dynamically indexing the directory's underlying storage allows us to achieve a 5000x speedup in single particle trajectory queries, which require reading all data for a single particle. This speedup increases with application scale while the overhead is fixed at 3% of available memory.

1 INTRODUCTION

Faster storage media, faster interconnection networks, and improvements in systems software have significantly mitigated the effect of I/O bottlenecks in HPC applications. Even so, applications that read and write data in small chunks are limited by the ability of both the hardware and the software to handle such workloads efficiently. Often, scientific applications partition their output using one file per process. This becomes a problem in HPC platforms such as Los Alamos National Laboratory's (LANL) Trinity supercomputer, which consists of more than 900,000 cores [5]. Moreover, this problem will worsen with exascale supercomputers, which are expected to be at least 25 times larger than Trinity [1]. To avoid wasting time to create output files on such machines, scientific applications are forced to use libraries that combine multiple I/O streams into a single file, such as HDF5 [21] and PLFS [9]. For many applications where output is produced out-of-order, this needs to be followed by a costly, massive data sorting operation. With DeltaFS, we allow applications to write to an arbitrarily large number of files, while also guaranteeing efficient data access without requiring sorting.

PDSW-DISCS'17, November 12–17, 2017, Denver, CO, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5134-8/17/11…$15.00
https://doi.org/10.1145/3149393.3149398

The first challenge when handling an arbitrarily large number of files is dealing with the resulting metadata load. We manage this using the DeltaFS file system [49]. DeltaFS extends prior work [34, 36, 48] by being *transient* and *server-less*. The transient property of DeltaFS allows each program that uses it to individually control the amount of computing resources dedicated to the file system, effectively scaling metadata performance under application control. When combined with DeltaFS's server-less nature, this metadata scaling approach allows file system design and provisioning decisions to be decoupled from the overall design of HPC platforms. As a result, applications that create one file for each process are no longer tied to the platform storage system's ability to handle metadata-heavy workloads. At the same time, the HPC platform can provide scalable file creation rates without requiring a fundamental redesign of the platform's storage system.

The second challenge, and the topic of this paper, is guaranteeing both fast writing and reading for workloads that consist primarily of small I/O transfers. This work was inspired by our interactions with cosmologists seeking to explore the trajectories of the highest energy particles in an astrophysics simulation using the VPIC plasma simulation code [13]. In each timestep of the VPIC simulation, a number of attributes are generated for every particle including its location in physical space and its magnetic energy. While the data produced for a single particle in a given timestep is small (tens of bytes), VPIC simulations often contain trillions of particles and run for tens of thousands of timesteps. Cosmologists may only identify the high energy particles at the end of the simulation, so all particle data needs to be retained until the end of the run. This results in an I/O pattern consisting of small write and read operations directed to a very large number of files, as we describe in Section 2.

To improve the performance of applications with small I/O access patterns similar to VPIC, we propose Indexed Massive Directory as a new technique for indexing data in-situ as data is written to storage. The goal of this in-situ data indexing function is to efficiently recall data that has been written to the same file without requiring any time-consuming data post-processing step to reorganize data. This greatly improves readback performance of applications downstream of VPIC in a workflow, at the price of small overheads associated with partitioning and indexing the data during writing. We present Indexed Massive Directories in more detail in Section 3.

Our experiments in Section 4 show that DeltaFS is able to speed up VPIC particle queries by many orders of magnitude. The indexing function itself consumes no more than 3% of the VPIC's memory, and the resulting indexes add a mere 8% overhead to the final output size. These results, achieved by applying software-defined design principles to HPC storage systems, promise improved application performance and simpler platform infrastructure in both system software stacks and dedicated hardware allocations.

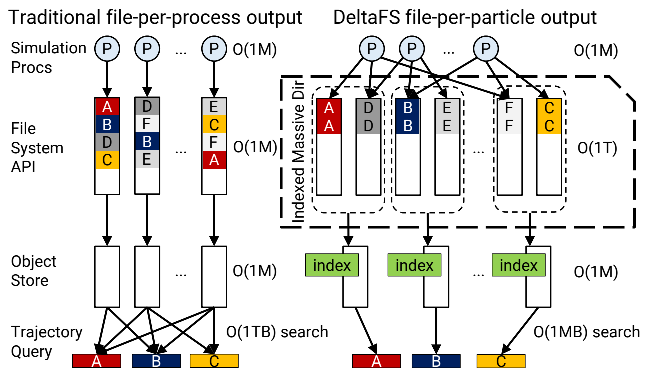


Figure 1: *DeltaFS in-situ indexing of particle data in an Indexed Massive Directory. While indexed particle data are exposed as one DeltaFS subfile per particle, they are stored as indexed log objects in the underlying storage.*

2 MOTIVATION

HPC platforms feature massive numbers of cores, huge memories, low-latency interconnects, and are typically backed by large scratch file systems for storing simulation output [10, 28]. As HPC clusters scale, these scratch file systems [16, 37, 38, 44] have become common performance bottlenecks for many parallel scientific applications. For an I/O bandwidth boost, newer HPC clusters use a *burst buffer*, i.e., an extra tier of storage with higher data access and transfer rates than the scratch file system, but significantly smaller in capacity [5, 10, 26, 29]. Despite having burst buffers, existing HPC storage is still considered slow by scientific application developers.

Scientific applications operate most efficiently when their run time consists mostly of computing, with very little time spent outputting simulation state to storage. Simulation state is periodically persisted to storage, in order to be later analyzed by the scientist, and to restart in case of failure. Applications output multiple files per process. Some applications may use libraries that combine multiple I/O streams into a single file [9, 21] to reduce the metadata overhead to the underlying storage. In order to quickly persist data to storage, however, the data is often written out of order, therefore increasing the latency of data queries after the simulation. Today, scientists typically resolve this issue through a massive data sorting operation, wasting valuable compute time that could be used to perform additional scientific work.

One such application is VPIC [13, 14], a highly-optimized particle simulation code developed at LANL. Each VPIC simulation proceeds in timesteps, and each process represents a bounding box in the physical simulation space that particles move through. Every few timesteps the simulation stops, and each process creates a file and writes the data for the particles that are currently located within its bounding box. This is the default, *file-per-process* mode of VPIC. For each timestep, 40 bytes of data is produced per particle representing the particle's spatial location, velocity, energy, etc. We refer to the entire particle data written at the same timestep as a *frame*, because frame data is often used by domain scientists to construct false-color movies of the simulation state over time. Large-scale VPIC simulations have been conducted with up to trillions of particles, generating terabytes of data for each frame.

Domain scientists are often interested in a tiny subset of particles with specific characteristics, such as high energy, that is not known until the simulation ends. All data for each such particle is gathered for further analysis, such as visualizing its trajectory through space over time. Unfortunately, particle data within a frame is written out of order, since output order depends on the particles' spatial location. Therefore, in order to locate individual particles' data over time, all output data must be sorted before they can be analyzed.

For scientists working with VPIC, it would be significantly easier programmatically to create a separate file for each particle, and append a 40-byte data record on each timestep. This would reduce analysis queries to sequentially reading the contents of a tiny number of particle files. Attempting to do this in today's parallel file systems, however, would be disastrous for performance. Expecting existing HPC storage stacks and file systems to adapt to scientific needs such as this one, however, is lunacy. Parallel file systems [37, 38, 43, 44] are designed to be long-running, robust services that work across applications. They are typically kernel resident, mainly developed to manage the hardware, and primarily optimized for large sequential data access. DeltaFS aims to provide this *file-per-particle* representation to applications, while ensuring that storage hardware is utilized to its full performance potential. A comparison of the file-per-process (current state-of-the-art) and file-per-particle (DeltaFS) representations is shown in Figure 1.

3 SYSTEM OVERVIEW

DeltaFS is a user-level file system that runs inside each simulation process on compute nodes [49]. It provides the simulation application a transient namespace decoupled from other applications running on the same cluster. To ensure high metadata throughput, DeltaFS packs directory entries, file inodes, and small files into large immutable objects (SSTable sets [3, 17]) stored in an underlying storage system [34]. Larger files are directly mapped to one or more data objects, fully utilizing the underlying storage bandwidth [23].

DeltaFS, with its efficient and scalable metadata path that enables high file creation rates [36, 48], seems like a natural fit to the file-per-particle model. With DeltaFS, each particle file is created under a shared DeltaFS directory. The file data and metadata within the directory are then packed into large SSTable objects, and the directory is split as necessary to utilize all DeltaFS instances running inside the parallel application. As with any other file system, however, using DeltaFS to access the metadata of trillions of files would cause too many I/O operations to the underlying storage. In the case of DeltaFS each I/O would be sequential and larger than for traditional file systems. Still, a large number of SSTables would need to be compacted in the background, as DeltaFS attempts to conserve buffer memory in order to avoid taxing the application.

Fortunately for DeltaFS, treating each particle as a full-fledged file (with its own file inode, directory entry, and other metadata) is largely unnecessary. In our case, particle files will only be used as logical containers to group data that is intended to be read together but may be written out-of-order. These applications have no interest in customizing the permissions, dates, or other attributes of individual files. Furthermore, renaming or removing individual files in this setting is also unusual. Since DeltaFS is created as a

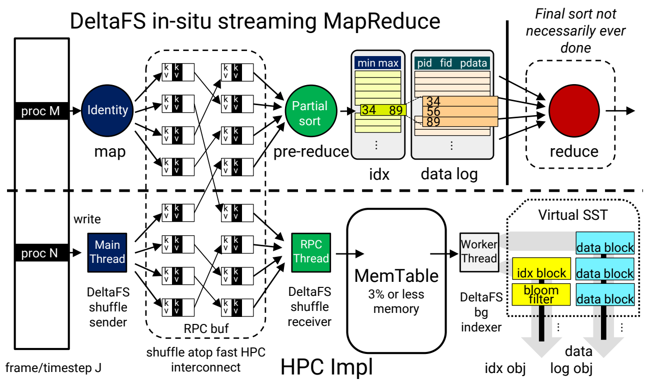


Figure 2: *A MapReduce perspective on DeltaFS Indexed Massive Directory serving an on-going VPIC simulation. Each record* `key` *consists of the frame ID (*`fid`*) and particle ID (*`pid`*). The particle data (*`pdata`*) make up the* `value`*. Records are shuffled by* `pid`*. Each* ***pre-reducer*** *does partial sort and indexing, and appends the resulting data, index, and filter blocks to two output objects to form* ***virtual SSTables*** *(SSTables not organized as one-file-per-table).*

user-space transient file system, it is able to customize a virtual file-per-particle representation for a simulation application.

As illustrated in Figure 1, when operating in a specially-marked directory, data written to the same filename are considered to belong to the same index key, and will be dynamically indexed by DeltaFS. Files created for each particle are implemented by DeltaFS as special inode-less files that we refer to as *subfiles*. Each subfile shares the inode of its parent directory to allow normal file system operations such as `open` and `close`. DeltaFS disallows `rename`, `chown`, `unlink`, as well as a few other operations on subfiles due to the absence of inodes and directory entries. In order to quickly read each subfile without first sorting the entire directory, the data of these subfiles are shuffled, indexed, and reorganized by DeltaFS as indexed log objects stored in the underlying storage. We refer to these specially-marked directories as *Indexed Massive Directories*.

An application creates a massive directory by calling `mkdir` with a special flag. To access each subfile inside a massive directory, an application first performs an `open` on the directory to obtain a directory handle. Then, the returned handle is used to perform an `openat` on the subfile, which returns a file handle that can then be used to perform normal `read` and `write` operations. The reason a massive directory must first be opened before any of its subfiles may be accessed is because DeltaFS does not allocate inodes or directory entries for subfiles, so the directory context is needed for DeltaFS to locate subfile data, as well as the indexes for those data.

In the remainder of this section we present the design of this in-situ data indexing function as a streaming MapReduce pipeline, highlight some of the techniques we used to overcome the restrictions and challenges for running an I/O service in-situ within an HPC application, and discuss its implementation.

A streaming MapReduce analogy. Consider a classic MapReduce [20] program that converts an existing VPIC simulation output from its original file-per-process representation to the new file-per-particle representation noted above. One way to achieve this would be creating a mapper for each VPIC output file and having that mapper generate a record for each 40-byte particle data read from the output file. In this case, each record would be a key-value pair. The key for these records should be encoded as a particle ID prefixed with a frame index, and the value would just be the original 40-byte particle data. Records generated by mappers would be shuffled and sent to reducers. This can be achieved by hashing particle IDs to a reasonable number of reducer processes (perhaps the same as the number of processes used for the original simulation). To make this efficient it would also be necessary to batch and shuffle together many particle records to fully utilize the network bandwidth.

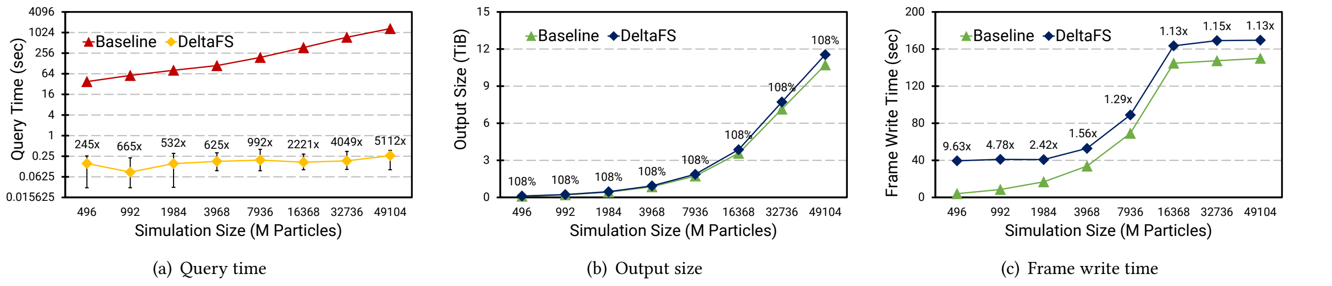
Because of the shuffling, reducers would see non-overlapping sets of particles. Assuming the sort before each reducer was already keyed by the frame index, each reducer could just perform a no-op and directly append each particle data to its final per-file destination. Unfortunately, allocating a destination file for each particle would be too costly for the underlying storage. A more realistic approach for each reducer is to append all data to a single per-reducer log file and to have a separate per-reducer index file to map every particle to its data locations. These data locations would be offsets within the log file that stored the relevant particle data.

Because particles were shuffled before reaching a reducer, a reader seeking a particle trajectory could just hash the ID of the particle to identify the reducer, and then read the index file produced by that reducer to locate all trajectory data. The size of the index file would be bounded in this case and would not increase with simulation scale. In addition, by separating the indexes from the data log generated by each reducer, these indexes could be fetched efficiently with large sequential reads.

General-purpose MapReduce engines like Hadoop [2] or Spark [4] could execute this, but would be extremely inefficient. This is because at HPC scale the temporary files created for the sorting before the reduce phase would not fit in the main memory of the compute nodes, and because most compute node memory would belong to the simulation application. With a tiny bit of memory, partially sorted data would have to be pushed to and from storage repeatedly until the final sort could ever be completed.

To avoid this inefficiency, reducers in the MapReduce pipeline must live with partially-sorted data so the final sort could be bypassed. We call these special reducers *pre-reducers*. The data sorted in each partial-sort is bounded by the memory DeltaFS is allowed to steal from the application. Each partial-sort produces a small sorted *run* of data. But without a final sort to merge all those runs, these runs will have overlapping ranges and a reader will have to check every run produced by a reducer in order to fetch a complete trajectory. To avoid potentially many unnecessary seeks to the underlying storage for non-existing data, a bloom filter [11] is created for each run and is appended to the per-reducer index file. A reader may use these bloom filters to perform membership tests and bound the number of storage seeks per trajectory per frame.

Figure 2 shows a logic MapReduce perspective on DeltaFS in-situ indexing. Shuffle sender (mapper) and receiver (pre-reducer) illustrated in the figure are merely roles taken by DeltaFS embedded inside each simulation process on compute nodes. No additional nodes or processes are needed to execute the pipeline. Each DeltaFS instance acts as a sender and a receiver simultaneously, and different DeltaFS instances communicate with each other through an RPC-based mechanism [15, 39]. Frame output written by the simulation code into DeltaFS will be converted into key-value pairs and



(a) Query time (b) Output size (c) Frame write time

Figure 3: *Results from real VPIC simulation runs with and without DeltaFS at LANL Trinity computer. Simulation and output experiments (c) are not variable, and we show the average of two runs, while query times (a) are much more variable, and we show the average of 100 different particles queried (all with cold caches).*

sent to bins (*MemTables*) allocated by DeltaFS at the receiver side using only a few percent (typically 3% or less) of the application's memory. When one of these bins is full, it will be sorted and indexed by a DeltaFS background thread. The sorted data is formatted as SSTable data blocks and is appended to that bin's log object in the underlying storage. The much smaller index for that data, as well as the bloom filter, is delayed longer, but eventually appended to that bin's index object as SSTable index and filter blocks. To ensure durability, frame data buffered in bins are forcefully flushed to storage at the beginning of the next frame output, when each shuffle receiver must have received all data for the previous frame.

4 EXPERIMENTS

To evaluate the performance of DeltaFS Indexed Massive Directories, we slightly modified VPIC to support our file-per-particle model. Specifically, rather than writing a tuple of particle ID and payload to a single output file per VPIC process, we write the payload to a file named by that particle's ID. We used *LD_PRELOAD* to redirect VPIC's file I/O to DeltaFS. Applications other than VPIC and regular tools may also use *LD_PRELOAD* to use DeltaFS.

Our experiments were performed on LANL's Trinity system [5]. Each Trinity compute node has 32 CPU cores and 128GiB DDR4 memory. These compute nodes are able to communicate with a small set of burst-buffer nodes through a dragon-fly network [7]. Each burst-buffer node has a peak data bandwidth of 5.33GiB/s, and provides about 5.83TiB of usable storage. In our experiments, we used a maximum of 99 compute nodes and 3 burst-buffer nodes.

We configured VPIC to simulate a fixed number of timesteps and then write 40 bytes of state per particle to the burst buffer. Each our experiment consisted of 5 such particle dumps. On Trinity each compute phase takes around 7 minutes to run. Once a simulation completes, we stage the output data from burst buffer to a backend Lustre file system as the burst buffer storage must be released at the end of each job. We then measure the performance of trajectory reads on a set of randomly selected VPIC particles. All reads go to the Lustre file system, because the output is no longer in the burst buffer. This models a post-processing phase at a later time.

We compare the performance of running VPIC both with and without DeltaFS. For the baseline case of VPIC without DeltaFS, the VPIC simulation writes one output file per process. For the VPIC with DeltaFS enabled, the simulation writes one DeltaFS file per particle. In the DeltaFS case, particle data will be dynamically indexed and stored as virtual SSTables. Post-processing trajectory reads from the Lustre file system, in the baseline case, are implemented by scanning the entire simulation output in parallel using the same number of CPU cores as the original simulation. DeltaFS trajectory reads, on the other hand, require using only a single CPU core to read the corresponding DeltaFS particle files. The data of these particle files can be quickly located by DeltaFS using the indexes it creates as it writes the data to the underlying storage.

We run VPIC at 8 different scales, simulating 496 million to 48 billion particles. We maintain a fixed ratio of 16 million particles per VPIC process and 31 VPIC processes per compute node. We reserve 1 core on each node for core specialization – specifically, we have found this is required to reduce network performance variance when using TCP communication over the Aries interconnect. Core specialization is enforced by SLURM via Linux `cgroups`. Thus, our smallest run uses 1 node with 31 processes on it, while our largest run uses 3069 VPIC processes on 99 nodes. We allocate one burst buffer node for every 16 billion particles. We present three types of experimental results: the time it takes to read the trajectory of a single particle from the simulation output (query time), the total size of data produced by the simulation (output size), as well as the total time it takes to write out all the particles at a specific timestep (frame write time).

Query time. In Figure 3(a), we show the time it takes to read the trajectory of a random particle for both the baseline runs and the DeltaFS runs. Note that both axes are shown in logarithmic scale. For the baseline runs, the particle query time is proportional to the simulation size. This is because we have no pre-computed indexes to map a particle to its physical data locations and thus have to scan the entire simulation output. This scan time quickly rises from 40 seconds for small runs up to 20 minutes for larger runs. On the other hand, DeltaFS is able to resolve trajectory queries in a small and bounded amount of time because the latency of reading a particle trajectory in DeltaFS is largely dependent on the size of the indexes that must be read into the memory. This time is tiny in our experiments due to partitioning, and it increases very slowly as simulation size increases. The plot shows the average query times for DeltaFS runs range from 150ms in the 496-million-particle case (a 245x improvement over the baseline), to 260ms in the 48-billion-particle case (a 5112x improvement over the baseline). Also note that all DeltaFS queries executed entirely on a single CPU core, while the trajectory queries for the baseline case use up to 3069 CPU cores for parallel data scanning.

Output size. Figure 3(b) shows the total output size produced by both types of runs. DeltaFS has the additional overhead of storing the indexes and filters it generates, plus it also has additional overhead from encoding and padding the simulation output and indexes. The figure shows that DeltaFS's additional overhead is moderate across our experiments (around 8%).

Frame write time. Figure 3(c) shows the average time it takes to write out a simulation frame to burst-buffer for both types of runs. The DeltaFS frame write time includes the additional cost of building and writing the in-situ indexing for particle data. Also note that in the first 5 runs of the plot only a single burst-buffer node was used. In the largest 3 runs more burst-buffer nodes were used (we fixed the ratio to 33 compute nodes to 1 burst-buffer node). For the first 5 runs, the figure shows DeltaFS has large but decreasing overheads. This is because those jobs are not large enough to saturate the burst-buffer node, so the system is dominated by the extra work DeltaFS does for the index. In the 3 larger runs, the system bottlenecks on burst-buffer bandwidth, and we start to see a converged DeltaFS slowdown of about 15%.

5 RELATED WORK

One important goal of DeltaFS is to replace traditional site-managed file system deployments [16, 22, 37, 38, 43, 44] with a transient and server-less file system service. The idea of a transient, server-less file system originates from our past work on scalable file system metadata [32, 36, 48, 49]. DeltaFS uses a modified LSM-Tree [31] scheme to pack file and directory metadata into large data objects [34, 35]. The initial design of our Indexed Massive Directory were inspired by PLFS [9, 12, 19] and MDHIM [24]

Server-less file systems [8, 37] are traditionally characterized by a set of symmetric file servers that are each capable of serving the entire file system namespace. This architecture is reused by BatchFS [48] to enable flexible metadata migration and service allocation, and is extended by DeltaFS [49] to be literally server-less.

The idea of leveraging application-owned resources can also be applied to data operations, such as storing application data directly on compute nodes using their local storage [46], or buffering checkpointing data inside the local memory of each compute node [33] or on the management of burst-buffer storage [6, 29, 40, 41].

Indexing is a popular means to improve the access performance for scientific data. ADIOS leveraged parallel indexing scheme to improve read performance for the Pixie3D simulation code [27, 30]. The indexes are constructed using FastQuery [18], a parallelized compressed bitmap index similar to the compressed bitmap indexing described by FastBit [45]. These techniques were then used to perform scalable scientific data index creation and in-situ data analysis. Our goal with DeltaFS similarly indexes VPIC particle data in parallel; however, DeltaFS provides the streaming MapReduce derived shuffle to deal with the large degree of entropy in VPIC particle data and uses bloom filters [11] for quickly building candidate sets of SSTables.

Multiple projects, including PreDatA [47], SMART [42], and GLEAN [25], have explored the use of embedded MapReduce pipelines for in-situ data analysis. These projects primarily used semantic knowledge about the data to perform indexing and analysis (whether in-situ or in-transit). DeltaFS differs both in its focus on constraining the resources used for shuffling and indexing, and that the MapReduce pipeline exists within the file system, is available via a POSIX interface, and performs indexing without using semantic knowledge of the data streaming into the file system.

Byna et al. performed petascale particle simulations using VPIC for trillions of particles in [13] and [14]. In [14], the authors experimented with two trillion particles and 2000 timesteps of simulation to produce about 350 TBs of data (including checkpoints). Despite abandoning the file-per-process (fpp) model and switching to a shared HDF5 model, the authors achieved comparable Lustre throughput.

6 CONCLUSION

DeltaFS is a transient, user-level file system that uses a software-defined resource allocation to customize available hardware to the needs of the application at hand. In our past work we have demonstrated how this approach results in scalable metadata performance for scientific applications at extreme scale. In this paper we show our implementation of Indexed Massive Directory as a special directory type for DeltaFS. By embedding a stripped-down streaming MapReduce pipeline into the POSIX write path of DeltaFS, we are able to transparently index massive numbers of files written to a single directory, and store them as indexed log objects.

We have evaluated the efficiency of this special directory type on Los Alamos National Lab's Trinity hardware. By applying in-situ partial sorting of VPIC's particle output, we demonstrated over 5000x speedup in reading a single particle's trajectory from a 48-billion particle simulation output using only a single CPU core, compared to post-processing the entire dataset (10TiB) using the same amount of CPU cores as the original simulation. This speedup increases with simulation scale, while the total memory used for partial sort is fixed at 3% of the memory available to the simulation code. The cost of this read acceleration is the increased work in the in-situ pipeline and the additional storage capacity dedicated to storing the indexes. We are working on further optimizing our techniques, nevertheless in their current state, they already demonstrate reasonable simulation slowdown during output phases (only 15% slower), which results in a total simulation slowdown of just about 5%. These results are encouraging, as they indicate that the output write buffering stage of the software-defined storage stack can be leveraged for one or more forms of efficient in-situ analysis, and can be applied to more kinds of query workloads.

ACKNOWLEDGMENTS

The authors would like to thank Phil Carns, Jerome Soumagne, Shane Snyder, and Robert Ross for their guidance on using the *Mochi* software stack (http://press3.mcs.anl.gov/mochi/), as well as David Nystrom and Cornell Wright for their advice on the use of LANL software and hardware. This material is based on work supported in part by the US DOE and Los Alamos National Laboratory, under contract number DE-AC52-06NA25396 subcontract 394903 (IRHPIT), and by the US DOE, Office of Science, Advanced Scientific Computing Research (ASCR) under award number DE-SC0015234. We also thank the member companies of the PDL Consortium (Broadcom, Citadel, Dell EMC, Google, Hewlett-Packard, Hitachi, Intel, Microsoft, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate, Two Sigma, Toshiba, Veritas, and Western Digital).

REFERENCES

[1] Exascale computing project. http://www.exascale.org.
[2] Hadoop. http://hadoop.apache.org/.
[3] Leveldb. https://github.com/google/leveldb/.
[4] Spark. https://spark.apache.org/.
[5] Trinity. http://www.lanl.gov/projects/trinity/.
[6] Ali, N., Carns, P., Iskra, K., Kimpe, D., Lang, S., Latham, R., Ross, R., Ward, L., and Sadayappan, P. Scalable I/O forwarding framework for high-performance computing systems. In *Proceedings of the 2009 IEEE International Conference on Cluster Computing (CLUSTER 09)*, pp. 1–10.
[7] Alverson, B., Froese, E., Kaplan, L., and Roweth, D. Cray xc series network. Tech. Rep. WP-Aries01-1112, Cray Inc., Nov. 2012.
[8] Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S., and Wang, R. Y. Serverless network file systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP 95)*, pp. 109–126.
[9] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., and Wingate, M. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 09)*, pp. 21:1–21:12.
[10] Bent, J., Settlemyer, B., and Grider, G. Serving data to the lunatic fringe: The evolution of HPC storage. *USENIX ;login: 41*, 2 (June 2016).
[11] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (July 1970), 422–426.
[12] Bonnie, D. J., and Torres, A. G. Small file aggregation with plfs. Tech. rep., Los Alamos National Laboratory, 2013.
[13] Byna, S., Sisneros, R., Chadalavada, K., and Koziol, Q. Tuning parallel i/o on blue waters for writing 10 trillion particles. In *Cray User Group (CUG)* (2015).
[14] Byna, S., Uselton, A., Prabhat, D. K., and He, Y. Trillion particles, 120,000 cores, and 350 tbs: Lessons learned from a hero i/o run on hopper. In *Cray User Group (CUG)* (2013).
[15] Carns, P., Ligon, W., Ross, R., and Wyckoff, P. Bmi: a network abstraction layer for parallel i/o. In *19th IEEE International Parallel and Distributed Processing Symposium* (April 2005).
[16] Carns, P. H., Ligon, W. B., Ross, R. B., and Thakur, R. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th USENIX Annual Linux showcase and Conference (ALS 00)*, pp. 317–328.
[17] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*, pp. 205–218.
[18] Chou, J., Wu, K., and Prabhat. FastQuery: A parallel indexing system for scientific data. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER 11)*, pp. 455–464.
[19] Cranor, C., Polte, M., and Gibson, G. Structuring PLFS for extensibility. In *Proceedings of the 8th Parallel Data Storage Workshop (PDSW 13)*, pp. 20–26.
[20] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Opearting Systems Design and Implementation (OSDI 04)*, pp. 10–10.
[21] Folk, M., Cheng, A., and Yates, K. Hdf5: A file format and i/o library for high performance computing applications. In *Proceedings of Supercomputing* (1999), vol. 99, pp. 5–33.
[22] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP 03)*, pp. 29–43.
[23] Gibson, G. A., Nagle, D. F., Amiri, K., Butler, J., Chang, F. W., Gobioff, H., Hardin, C., Riedel, E., Rochberg, D., and Zelenka, J. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 98)*, pp. 92–103.
[24] Greenberg, H. N., Bent, J., and Grider, G. MDHIM: A parallel key/value framework for HPC. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage 15)*, pp. 10–10.
[25] Hereld, M., Papka, M. E., and Vishwanath, V. Toward simulation-time data analysis and i/o acceleration on leadership-class systems. In *Proc. IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV2011)* (Providence, RI, 10/2011 2011).
[26] Inman, J., Vining, W., Ransom, G., and Grider, G. MarFS, a Near-POSIX interface to cloud objects. *USENIX ;login: 42*, 1 (Jan. 2017).
[27] Kim, J., Abbasi, H., Chacón, L., Docan, C., Klasky, S., Liu, Q., Podhorszki, N., Shoshani, A., and Wu, K. Parallel in situ indexing for data-intensive computing. In *Proceedings of the 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV 11)*, pp. 65–72.
[28] LANL, NERSC, and SNL. Apex workflows. Tech. rep., Los Alamos National Laboratory (LANL), National Energy Research Scientific Computing Center (NERSC), Sandia National Laboratory (SNL), Mar. 2016.
[29] Liu, N., Cope, J., Carns, P., Carothers, C., Ross, R., Grider, G., Crume, A., and Maltzahn, C. On the role of burst buffers in leadership-class storage systems. In *Proceedings of the 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST 12)*, pp. 1–11.
[30] Liu, Q., Logan, J., Tian, Y., Abbasi, H., Podhorszki, N., Choi, J. Y., Klasky, S., Tchoua, R., Lofstead, J., Oldfield, R., Parashar, M., Samatova, N., Schwan, K., Shoshani, A., Wolf, M., Wu, K., and Yu, W. Hello ADIOS: The challenges and lessons of developing leadership class I/O frameworks. *Concurr. Comput. : Pract. Exper. 26*, 7 (May 2014), 1453–1473.
[31] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. The log-structured merge-tree (LSM-tree). *Acta Inf. 33*, 4 (June 1996), 351–385.
[32] Patil, S., and Gibson, G. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST 11)*, pp. 13–13.
[33] Rajachandrasekar, R., Moody, A., Mohror, K., and Panda, D. K. D. A 1 PB/s file system to checkpoint three million MPI tasks. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing (HPDC 13)*, pp. 143–154.
[34] Ren, K., and Gibson, G. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pp. 145–156.
[35] Ren, K., Zheng, Q., Arulraj, J., and Gibson, G. Slimdb: A space-efficient key-value storage engine for semi-sorted data. *Proc. VLDB Endow. 10*, 13 (Sept. 2017), 2037–2048.
[36] Ren, K., Zheng, Q., Patil, S., and Gibson, G. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 14)*, pp. 237–248.
[37] Schmuck, F. B., and Haskin, R. L. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, pp. 231–244.
[38] Schwan, P. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Ottawa Linux Symposium (OLS 03)*, pp. 380–386.
[39] Soumagne, J., Kimpe, D., Zounmevo, J., Chaarawi, M., Koziol, Q., Afsahi, A., and Ross, R. Mercury: Enabling remote procedure call for high-performance computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)* (Sept 2013), pp. 1–8.
[40] Wang, T., Mohror, K., Moody, A., Sato, K., and Yu, W. An ephemeral burst-buffer file system for scientific applications. In *Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 16)*, pp. 69:1–69:12.
[41] Wang, T., Moody, A., Zhu, Y., Mohror, K., Sato, K., Islam, T., and Yu, W. Metakv: A key-value store for metadata management of distributed burst buffers. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (May 2017), pp. 1174–1183.
[42] Wang, Y., Agrawal, G., Bicer, T., and Jiang, W. Smart: A mapreduce-like framework for in-situ scientific analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2015), SC '15, ACM, pp. 51:1–51:12.
[43] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*, pp. 307–320.
[44] Welch, B., Unangst, M., Abbasi, Z., Gibson, G., Mueller, B., Small, J., Zelenka, J., and Zhou, B. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 08)*, pp. 2:1–2:17.
[45] Wu, K. Fastbit: an efficient indexing technology for accelerating data-intensive science. *Journal of Physics: Conference Series 16*, 1 (2005), 556.
[46] Zhao, D., Zhang, Z., Zhou, X., Li, T., Wang, K., Kimpe, D., Carns, P., Ross, R., and Raicu, I. FusionFS: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In *Proceedings of the 2014 IEEE International Conference on Big Data (BigData 14)*, pp. 61–70.
[47] Zheng, F., Abbasi, H., Docan, C., Lofstead, J., Liu, Q., Klasky, S., Parashar, M., Podhorszki, N., Schwan, K., and Wolf, M. PreDatA - preparatory data analytics on peta-scale machines. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010* (05 2010), pp. 1 – 12.
[48] Zheng, Q., Ren, K., and Gibson, G. BatchFS: Scaling the file system control plane with client-funded metadata servers. In *Proceedings of the 9th Parallel Data Storage Workshop (PDSW 14)*, pp. 1–6.
[49] Zheng, Q., Ren, K., Gibson, G., Settlemyer, B. W., and Grider, G. DeltaFS: Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop (PDSW 15)*, pp. 1–6.