June 14, 2019
Ver. 34
DRAFT

Thesis Proposal

Distributed Metadata and Streaming Data Indexing as Scalable Filesystem Services

Qing Zheng

2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Garth Gibson, Co-Chair
George Amvrosiadis, Co-Chair
Gregory Ganger,
Bradley Settlemyer, Los Alamos National Laboratory

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This thesis is partially supported by the Los Alamos National Lab's Institute for Reliable High Performance Information Technology (IRHPIT), the U.S. Dept of Energy's Advanced Scientific Computing Research (ASCR) program, the U.S. National Science Foundation's Parallel Reconfigurable Observational Environment (PRObE) project, the New Mexico Consortium's Ultra System Research Center (USRC), and the PDL Consortium.

Thesis Statement

Deep relaxation of filesystem metadata synchronization and serialization improves the scalability of HPC filesystems. Software-defined filesystem directory types are effective mechanisms for data to be more flexibly stored and indexed without requiring the underlying storage to understand all data structures and without requiring expensive post-processing.

1 Introduction

Scientific applications are typically highly-optimized parallel simulation programs that run on High Performance Computing (HPC) platforms. A notable difference separating HPC platforms from commodity computing platforms is the efficiency the former possesses for running a single large application that consumes the entire machine. Similarly, these platforms' underlying filesystems, a special class of filesystem known as the parallel filesystem [15, 70, 72, 84], are uniquely optimized for concurrent accesses to a single file or a single directory. Unlike many commodity filesystems that are paired with co-located compute and storage for cost-effective high bandwidth [19, 24], HPC filesystems are typically deployed with dedicated storage. This better isolates the failure domain of compute from that of storage [75], and helps improve system manageability with storage nodes each having its own operating system (OS), and data more conveniently shared across multiple computing platforms [57]. This thesis targets large-scale HPC applications that use tens of thousands of nodes across hundreds of thousands of CPU cores.

1.1 A Two-Pronged Approach for Overcoming HPC Bottlenecks

As people build larger and more powerful supercomputers [2, 22, 40, 58], we are standing at the dawn of the exascale age. The sheer size of future machines will bring unprecedented levels of concurrency. For applications that write one file per process, increased concurrency will cause more files to be accessed simultaneously and this requires the metadata information of these files to be managed more efficiently [64, 66]. An important factor preventing existing HPC filesystems from being able to more efficiently absorb filesystem metadata mutations is the continued use of a single, globally consistent filesystem namespace to serve all applications running on a single computing environment. Although having a shared filesystem namespace accessible from anywhere in a computing environment has many welcome benefits [27, 50, 59], it increases each application process' communication with the filesystem's metadata servers for synchronizing and serializing concurrent filesystem metadata changes. While modern parallel filesystems separate metadata from data movement [25], excessive filesystem metadata synchronization and serialization activities can still noticeably hamper job progress and are considered harmful in general [1, 16]. This is especially the case when all the metadata synchronization and serialization work is coordinated by a small, fixed set of filesystem metadata servers as we see in many HPC platforms today [31, 70, 72]. Since scientific applications are typically self-coordinated batch programs, the **first theme of this thesis** is about taking advantage of knowledge about the system and scientific applications to drastically reduce, and in extreme cases remove, unnecessary filesystem metadata synchronization and serialization, enabling HPC applications to better enjoy the increasing level of concurrency in future HPC platforms.

While overcoming filesystem metadata bottlenecks during simulation I/O is important, achieving efficient analysis of large-scale simulation output is an even more important enabler for fast scientific discovery. With future exascale machines, simulation output will only become larger and more detailed than it is today. Some modern-day scientific data analysis already consists of queries that are highly selective and these types of queries will continue to be important [14, 77]. To prevent analysis queries from experiencing excessive I/O delays, a simulation's output can be, at the cost of favoring sparse queries, carefully reorganized for efficient retrieval. Data reorganization is effective because simulation output is often written

without considering the efficiency of the queries following the simulation [47, 76]. But data reorganization can be extremely time-consuming when its processing requires data to be read back from storage in large volumes [17, 20]. The **second theme of this thesis** is about leveraging idle CPU cycles during the writing of data on the compute nodes of an application to perform data reorganization and indexing, enabling data to be transformed to a read-optimized format without undergoing expensive readbacks.

1.2 Thesis Components

This thesis consists of three components. The first and the second thesis components are centered on developing new filesystem metadata designs with drastically relaxed filesystem metadata synchronization and serialization. The third component focuses on developing techniques for leveraging idle computing resources available during application I/O to serve efficient data reorganization and indexing.

Deep metadata writeback caching (§3.2). Scientific applications are typically carefully programmed simulation jobs that do not necessarily benefit from the strict semantics provided by traditional filesystems. For example, many scientific applications may not require sequential consistency or immediate global visibility for namespace updates [7]. Taking advantage of this property, **the first component of this thesis** presents two deep metadata writeback caching mechanisms, allowing a set of cooperative client processes to delay applying certain filesystem metadata mutations [66, 92]. That is, instead of synchronously communicating every filesystem metadata mutation to the server, each process can log some of its mutations in a private writeback cache, deferring all synchronization and serialization operations until an eventual bulk insertion, where all previously logged mutations are batch integrated to the global filesystem namespace. To achieve this efficiently, this thesis uses a log format derived from TableFS [64] in which each filesystem metadata mutation is recorded as a row in a set of tables constructed with a modified LevelDB realization of a Log-Structured Merge Tree (LSM-Tree) [56].

Two deep writeback caching mechanisms are developed. One is pessimistic and was demonstrated with IndexFS [66]. The other is optimistic and was demonstrated with BatchFS [92].

Fully-decentralized metadata (§3.3). By aggressively decreasing the frequency of metadata synchronization and serialization, deep metadata writeback caching enables a client process to quickly absorb a large number of filesystem metadata mutations. But to make its metadata changes useful to others, the process must synchronize with a dedicated filesystem metadata server to merge logged mutations. To further relax synchronization, **the second component of this thesis** goes beyond deep writeback caching and bulk insertion, and explores a case in which a user is allowed to more intensely delay the merge of its filesystem metadata mutations, possibly forever if, for example, all users of the data are known a priori. To achieve this, a bold filesystem metadata architecture is imagined in which filesystem namespaces are no longer deemed global or implicitly shared among all application processes, and clients' private writeback caches of filesystem metadata mutations are each considered a separate namespace. This keeps unrelated job processes from sharing a single filesystem namespace unnecessarily, or from having to witness each other's filesystem metadata mutations. Also relaxed in this bold design is the conventional use of dedicated server machines to serve filesystem metadata. Instead, a parallel job directly instantiates a filesystem metadata service in client middleware that operates on only scalable object storage and com-

municates with other jobs by sharing or publishing logs of filesystem metadata mutations. This results in a fully-decentralized filesystem metadata layer, and constitutes this thesis's most aggressive filesystem metadata synchronization relaxation. Because the core idea is to have different job processes see different filesystem namespaces, this thesis refers to this new design as *no ground truth*.

While having no ground truth may largely eliminate false sharing in the write path, it increases the work a filesystem has to do in the read path when the results of multiple previous jobs are combined and used as the input for a follow-up job. This includes detecting and resolving potential conflicts, and looking at a potentially large number of places in storage for relevant metadata information. To improve read responsiveness, instead of having each reader risk performing an excessive amount of work whenever a filesystem metadata read operation is executed, one or more namespace curation services can be deployed to have related filesystem metadata mutations pre-sorted and indexed in memory such that subsequent filesystem metadata read operations can complete faster.

The idea of a no-ground-truth parallel filesystem is still on-going work. Benefits of this design were partially demonstrated with DeltaFS [93].

Dynamic Data Reorganization and Indexing (§3.4). Many scientific simulations are written as bulk synchronous parallel programs [43]. They generate output by periodically halting computation and writing information to storage. With modern HPC platforms, this process is expected to be slow and blocked on storage. This leaves idle CPU cycles on the compute nodes of a simulation application. After a simulation run, fast data analysis is traditionally achieved by careful data post-processing. But due to the growing size of data, data post-processing is becoming increasingly time-consuming as it often requires data to be read back from storage in large volumes. To improve time-to-insight, **the last component of this thesis** aims to overlap data reorganization and indexing with simulation I/O so that data can be optimized for fast queries without incurring a large number of data readbacks.

To reduce overhead, this thesis leverages the aforementioned idle CPU cycles to perform data reorganization and indexing. Processing data in parallel with simulation I/O across a large number of CPU cores enables the dynamic transformation of data representation. Dynamically transforming data as it streams to storage further enables expensive data readbacks to be avoided. The overall process can be viewed as a special form of simulation-time data processing, or *in-situ* data processing. Dynamically transformed data allows post-simulation data analysis queries, or *post-analysis* queries, to be answered efficiently.

Dynamic data reorganization and indexing is developed as a special directory type called the Indexed Massive Directory in DeltaFS [90]. It was demonstrated on up to 131,072 CPU cores [91].

1.3 Work Done

- The **design** of two different deep filesystem metadata writeback caching mechanisms allowing a set of collaborative client processes to quickly absorb a large number of filesystem metadata mutations. One mechanism is pessimistic and uses locks. The other is optimistic, relaxing the use of locks. Both mechanisms improve performance by deferring until an eventual bulk insertion the synchronization and serialization of a burst of filesystem metadata mutations that are known to be cooperative (§3.2).

- A modified LevelDB **realization** of a LSM-Tree allowing fast bulk insertion operations. Fast LSM-Tree bulk insertion, along with a log-structured filesystem metadata representation, is key to enabling fast filesystem metadata bulk insertion. The latter is an integral component of a deep filesystem metadata writeback cache (§3.2).
- An **evaluation** of the two deep filesystem metadata writeback caching mechanisms using two prototype systems, IndexFS and BatchFS, respectively. Experiment results show that client-side deep writeback caches can significantly improve filesystem metadata throughput (e.g., file creates, deletes, and opens) when the metadata changes a client makes are known to be cooperative (§3.2).
- A **preliminary study** of a fully-decentralized filesystem metadata design featuring no global filesystem namespace, no dedicated filesystem metadata server, and no ground truth. Preliminary results from a prototype system, DeltaFS, show that decentralized, transient, per-job filesystem metadata servers scale better than, and beyond, a fixed set of dedicated filesystem metadata server machines (§3.3).
- The **design** of an in-situ data processing mechanism for dynamically reorganizing and indexing data as data is written to storage. This new approach improves traditional post-processing approaches by being able to transform data to a read-optimized format without requiring massive readbacks (§3.4).
- A **realization** of this in-situ data processing mechanism as a software-defined filesystem directory type that can be efficiently instantiated on client middleware. To achieve high performance, this realization includes techniques for scalable all-to-all communication among a large number of CPU cores, a new data partitioning scheme for fast data partitioning, and a flattened LSM-Tree for efficient indexing of bursts of scientific data (§3.4).
- A large-scale **demonstration** of this realization using a prototype system, DeltaFS, and a scalable particle simulation code, VPIC, on LANL's Trinity computing cluster with a shared scientific goal of performing trajectory analysis across a small subset of one trillion particles.

  The largest run used 131,072 CPU cores, 4,096 compute nodes, and simulated 2 trillions particles (i.e., 2 trillion keys). Results show that dynamically generated data indexes allow per-particle trajectories to be quickly recalled regardless of how large the simulation is (which was not considered possible without performing potentially costly data post-processing or in-transit data computing). In addition, producing data indexes as data is written to storage only slightly increases (10%-35%) write time and requires only a small amount of extra storage ($\sim$3%) for storing data indexes (§3.4).

1.4 Work To Be Done

- More **evaluation** of the two proposed deep filesystem metadata writeback cache mechanisms with a focus on the cost of deferred filesystem metadata synchronization and serialization as opposed to their benefits on the write path (§3.2).
- A complete **design** of a no-ground-truth parallel filesystem. This includes a new filesystem metadata representation, a mechanism for finding and merging filesystem namespace snapshots, a scheme for garbage collection, and a high-performance realization of this idea (§3.3).
- A **demonstration** of this no-ground-truth idea on real-world scientific applications and workflows. This includes a prototype no-ground-truth filesystem implementation, integration of this prototype with a

selective set of scientific applications and workflows, and a comparison of this new approach with traditional approaches that use dedicated filesystem metadata servers (§3.3).

Work is expected to be completed within 1 year of the proposal date.

2 Background and Related Work

This section starts by linking HPC fault-tolerance to filesystems' capability to handle the growing number of files. Next it discusses ways for filesystems to better handle the growing number of files. Then it discusses some of the major problems with dedicating server machines for running global filesystem namespaces. Following that, it discusses the current state-of-the-art for doing data post-processing and reorganization. Finally, it notes the software-defined storage principle embodied in this thesis.

2.1 HPC Fault-Tolerance and Filesystem Metadata Performance

A key reason an application's I/O increases with its problem size is fault tolerance [29, 71]. As hardware is not reliable, many scientific applications write checkpoints to shield themselves from various system failures [35, 69]. In cases where an application's checkpoint size increases with its problem size, increasing the problem size increases the application's I/O for checkpointing [6, 8]. And because an application is more likely to fail when its problem size increases, increasing the problem size additionally requires the application to perform checkpointing more frequently and this causes more I/O activities [68, 88].

For applications that write one file per process, increased I/O activities cause more files to be accessed simultaneously and this requires the metadata information of these files to be managed more efficiently [21, 60, 83, 84, 87]. Applications write one file per process because this is known to be a robust way to hope the filesystem distributes data more evenly across its backend storage servers [72]. This allows the underlying storage bandwidth to be more fully utilized so the application is able to minimize its time spent reading and writing data and maximize time spent on scientific calculation [61, 62]. Another reason applications write one file per process is to avoid concurrent write sharing. Removing such sharing reduces lock contention and minimizes partial block updates [7, 70, 72, 84]. An important reason partial block updates are inefficient is that each such update requires block data to be read from storage, updated in memory, and then written back to storage.

2.2 Better Handling the Growing Number of Files

The **first step** to better handle the growing number of files is to devise an efficient filesystem metadata representation that reduces the number of disk seeks required by each filesystem metadata mutation. This is accomplished by work done under SlimFS [65] and TableFS [64], with filesystem metadata stored as rows in tables constructed by a modified LevelDB's realization of a LSM-Tree [56].

This thesis continues to use LSM-Trees for indexing filesystem metadata. Not only are LSM-Trees more write-optimized than regular B-Trees, but they can also be extended to enable fast filesystem metadata bulk insertion operations and to implement efficient filesystem namespace snapshots [66, 92, 93].

The **second step** to better handle the growing number of files is to devise a better namespace partitioning scheme that allows more servers to handle filesystem metadata operations simultaneously. This is accomplished by work done under IndexFS [66] and ShardFS [86], with filesystem namespaces aggressively partitioned to all available filesystem metadata servers in both designs.

Unfortunately, even with an efficient filesystem metadata representation and an aggressive filesystem namespace partitioning scheme, today's filesystem metadata semantics still expect every filesystem metadata operation an application executes to be globally synchronized and serialized, and HPC applications continue to experience bottlenecks. As one example, a key problem faced by IndexFS and ShardFS is to correctly enforce access control over all pathname lookups performed in a hierarchical filesystem namespace. That is, access to an object specified by a pathname requires permission to lookup that object's name in its parent directory, and permission to lookup that parent directory's name in the grandparent directory, recursing as specified by the pathname back to either the filesystem's root directory or a directory currently open in the caller's process. This multi-lookup resolution of a pathname is conventionally regarded as an atomic operation that must be carefully synchronized (e.g., by obtaining locks from one or more filesystem metadata servers) and serialized (e.g., by waiting for existing locks to clear) with respect to all filesystem operations on pathnames, and metadata operations with a same prefix of their absolute pathnames may conflict with one another. Consequently, pathname resolution, being one of many filesystem metadata activities that require global synchronization and serialization, constitutes a main bottleneck for filesystem metadata scaling, especially when multiple filesystem metadata servers are used.

To optimize pathname lookups, IndexFS uses dynamically partitioned namespace with client caching whereas ShardFS explores replicated directories with sharded files as an alternative. But to really achieve the levels of scale and performance future platforms require, synchronization of anything global should be avoided as much as possible so parallel filesystem metadata accesses can be further decoupled and parallelized. This thesis builds upon prior work, but better leverages knowledge about the system and HPC applications to explore deeper degrees of decoupling than do TableFS, IndexFS, and ShardFS.

2.3 Filesystem Metadata Semantics and Their Relaxation

Today, every filesystem metadata operation invoked by any process in an HPC cluster is expected to be synchronized and serialized with respect to all metadata operations invoked by any other process running in the cluster. One reason filesystems choose such strict, online transaction processing (OLTP) like semantics for metadata is to secure instant global visibility, which allows filesystem metadata mutations completed at one client to be immediately accessible to all filesystem clients requesting new metadata information. This is a vital property in terms of applications that use filesystems to communicate. Another reason filesystems choose strict semantics for metadata is to allow filesystem namespace integrity (e.g., no filename collisions) to be promptly checked and defended. This is important for potential mistakes and errors to be propagated back to the original application call site at the earliest possible moment, which is

in turn crucial for applications to recover, or die, quickly. While historically developed from early single-node operating systems, semantics as such now largely represent what people believe as filesystems.

Synchronously executing and serializing every filesystem metadata operation causes frequent client-server communication, which can be prohibitively expensive at scales. To alleviate this problem, delegations are often leased to client processes that enable direct operation over disjoint portions of a filesystem's namespace. Upon lease expiration or invalidation, mutations handled by a client are replayed at the server.

Unfortunately, even though mechanisms as such help decouple and parallelize filesystem metadata accesses, they are typically carefully designed to preserve global filesystem metadata synchronization and serialization properties, and therefore can not provide enough decoupling and parallelism for future HPC applications running on exascale machines. For example, leased delegations are typically immediately invalidated when other clients access respective regions of a filesystem's namespace. Similarly, delegations with write accesses (e.g., a write lock on a directory) are typically only leased to a single client process, with concurrent write accesses (e.g., multiple processes create files under a single directory simultaneously) either causing leases to be rapidly transferred among multiple client processes, or causing all leases to be revoked so future accesses must directly synchronize with the server [59, 84]. It is possible for the use of more fine-grained delegations to further decouple and parallelize filesystem metadata accesses [21, 70], but so long as global filesystem metadata synchronization and serialization properties are withheld, marginal performance improvements as such will continue to be dwarfed by the increased concurrency seen in future exascale HPC platforms.

In database systems, optimistic concurrency control (OCC) is often elected when data contention is known to be rare, a characteristic that many HPC applications share as well. One way to achieve OCC is to allow transactions to each operate on a snapshot of a database, perform all operations in complete isolation, and then undergo a single verification step that checks all potential conflicts [37]. This allows transactions to complete without the expense of managing locks and without having transactions wait for other transactions' locks to clear. As a result, transactions can experience less coupling from each other and more transactions can make progress in parallel.

Inspired by the ideas of OCC, the goal of this thesis is to leverage fast filesystem metadata bulk insertion and fast formation of filesystem namespace snapshots to construct protocols to aggressively defer filesystem metadata synchronization and serialization, and in extreme cases, without requiring locks. That is, clients optimistically assume there will be no conflict, and do not do two-phase locking. And instead of synchronously integrating filesystem metadata mutations, each client is allowed to log mutations in a private writeback cache, until an eventual bulk insertion that batch commits all logged changes.

The cost of deferred synchronization and serialization is delayed error checking and hence a potential for conflicting updates. To efficiently deal with conflicts, however, this thesis takes advantage of an efficient filesystem metadata representation to allow conflicts to be quickly detected, reconciled, or in some cases, quarantined. And different from traditional transactional databases, this thesis allows a batch of client mutations to fail partially during a bulk insertion and does not necessarily roll back the whole "transaction". This avoids a minor infraction from destroying a potentially large amount of work. Notifications of rejected filesystem metadata modifications are available to users in external logs and associated files may be retained with mechanically modified names for users to resolve conflicts later.

2.4 Shifting Away from Global Filesystem Namespaces

While scientific applications are programmed to use the platform's underlying filesystem to access files, this underlying filesystem does not have to be a long-standing service that maintains a global filesystem namespace, and does not have to use only dedicated resources. In fact, an important factor restricting existing HPC platforms' metadata performance is the use of a shared filesystem metadata plane to provide a global filesystem namespace that serves all applications running on a single computing platform.

First, having a shared filesystem namespace increases each application's communication with the filesystem's metadata servers. This is due to the increased filesystem metadata cache invalidation and subsequent lease renewal traffic caused by filesystem metadata activities made by other concurrent applications running on the same platform. For more communication is needed to execute filesystem metadata operations, it limits the overall metadata throughput each application is able to effectively receive.

Second, dedicating a shared filesystem metadata plane increases the burden of achieving efficient resource allocation. This is because the total amount of machine resources devoted to each HPC platform is largely fixed and cannot be easily changed once the platform goes online, so potentially many machine resources will have to be reserved for the shared filesystem metadata plane just for it to be ready for an envisioned peak metadata demand. But since the right amount of resources can be difficult to estimate, sticking to a fixed set of dedicated server machines almost always causes a waste of the machines' computing resources when the demand is too low, or a bottleneck when the demand is too high.

2.5 Scientific Applications and Data Reorganization for Post-Analysis

Many scientific applications are time-based simulations that run in timesteps. In many scenarios simulation state is periodically saved to storage for post-analysis. For simulations whose state involves lots of small objects, the simulation state is most efficiently dumped when these small objects are batched together and appended to storage using large sequential writes [67]. To operate efficiently, applications minimize their time spent writing state to slow storage in order to maximize time spent in simulation.

For simulations whose output is not written in the optimal order for post-analysis queries, output data must be carefully reorganized so queries can be answered quickly [18, 20]. Unfortunately, despite the deployment of faster interconnection network and larger parallel filesystems, storage remains a primary bottleneck both for simulation I/O and for data reorganization [6, 8, 39]. Today, moving data in large volumes is still costly. And the ever-widening gap between compute and I/O is slowly increasing the overhead of traditional post-writing data reorganization approaches [17, 18]. As simulations keep gaining size and resolution, there is also an opportunity for the use of more optimized data structures to produce more efficient data indexes and storage layouts. While faster storage media are available, their effective bandwidth is limited by the interconnection speed, and their reduced storage density prevents data from being stored solely in such media [46, 48]. For future platforms, not only will scientific analysis continue to be slow and blocked on storage when data is not structured for the reads, but the cost of data reorganization itself will become increasingly prohibitive when it is performed inefficiently.

2.6 Different Approaches for Data Reorganization

Traditionally, data reorganization is done by waiting until all data produced by a simulation is written to storage and then having a separate post-processing program to readback the simulation's data from storage and then transform it to a more read-optimized format. While this form of data reorganization conveniently decouples a simulation from its final data representation, the expensive data readback step it requires renders it increasingly inefficient and time-consuming as the compute-I/O gap grows.

One way to drastically reduce data readback is to reorganize data as data streams to storage. This can be generally referred to as in-situ data processing. Current state-of-the-art achieves it by adding additional nodes to a job to stage data so data can be asynchronously reorganized while the original simulation switches to its own computation [4, 5, 53, 54, 78, 79, 89]. An important drawback to this approach is the extra job nodes that must be dedicated to perform the in-situ reorganization of the job's periodic output.

To improve resource utilization, this thesis explores the use of idle computing resources on the compute nodes of an application to complete in-situ data reorganization and indexing. This avoids the needs of additional job nodes. Idle computing resources are temporarily available because some scientific applications are effectively forced to pause their computation during their I/O phases [8]. This thesis refers to in-situ processing that leverages only those temporarily available computing resources as embedded in-situ processing.

2.7 Software-Defined Filesystem Software

An overarching principle of this thesis is the use of client middleware to redefine data and filesystem metadata accesses. As demonstrated by lots of previous work [7, 33, 63, 80], client middleware, written as user-space code, affords semantics and optimizations that are more tailored to the needs of the application at hand, and can be deployed without changing the rest of the platform. This enables filesystems to better adapt to new types of applications and computing environments, and to better accommodate the increasing level of concurrency to be seen in future exascale machines.

In addition, deploying filesystems as client middleware allows the computing resources on the main computing platform to be better utilized, and can help reduce server-side resource contention seen in traditional filesystem servers. Moreover, the compute nodes' interconnection network, typically faster than the storage system, can too be used by the filesystem to implement more efficient filesystem operations.

Finally, dynamically instantiating filesystems as client middleware can better decouple applications from the platform's underlying storage. This is because applications will have the ability to choose from a potentially wide variety of filesystem middleware implementations, and to self-decide the right amount of computing resources to be devoted to the filesystem. As a result, filesystem design and provisioning decisions can be separated from the overall design of the platform, and future HPC platforms can be built with only scalable object storage and no longer need to size the filesystem potentially prematurely.

3 Thesis Components

Section 3.1 discusses difficulties and assumptions. Section 3.2 discusses deep metadata writeback caching and its implementations in IndexFS and BatchFS respectively. Section 3.3 discusses filesystem metadata decentralization and its realization in DeltaFS. Finally, Section 3.4 discusses streaming data reorganization and its implementation in DeltaFS as a special type of directory named the Indexed Massive Directory.

3.1 General Assumptions for Scientific Applications

A) Self-coordination. Unlike interactive programs such as OS shells, many scientific applications are self-coordinated batch programs that, when started, continue until completion without requiring human intervention. To perform I/O operations, these applications typically allow filenames to be constructed during program execution, or directly use filenames supplied by users. To maximize efficiency, programmers typically configure or program their applications to avoid filename collisions. This convention makes it possible for a filesystem to improve performance by aggressively deferring the synchronization of certain filesystem metadata mutations that are known to be cooperative (e.g., concurrent unique file creates beneath a checkpointing directory). By exploiting this convention, deferred mutations can be efficiently applied later using a filesystem batch mechanism without risking accumulating a large number of errors (e.g., filename conflicts) that can be expensive to fix or wasting a potentially giant portion of work.

B) Sequential sharing. Scientific applications persist information by writing data into files that are dynamically created during an application run. As a side effect of the scheduling of work, these files are often not read until after the application has completed its write phase. This renders sequential consistency and immediate global visibility for namespace updates almost unnecessary for these files, making it possible and more efficient for a set of cooperative client processes to buffer per-process namespace mutations, and to form deep storage writeback caches to serve in-situ data reorganization and indexing. There is a special use case, however, for application owners monitoring their process' output files for user steering and fast detection of wasted resources. This thesis deals with this separately.

C) One file per process. As HPC best practice, scientific applications typically have their processes write data into separate files. Because different processes operate on different sets of files, it enables each process to establish a private filesystem metadata writeback cache that is decoupled from those of other processes. One file per process is often referred to as N-N parallel I/O. This thesis focuses on this parallel I/O pattern. N-1 parallel I/O, on the other hand, can be efficiently transformed into N-N I/O with middleware software such as the PLFS filesystem [7].

3.2 Deep Metadata Writeback Caching

To maximize resource utilization, scientific applications are typically carefully programmed to perform filesystem operations cooperatively. This reduces filesystem data and metadata contentions. For example, many parallel scientific applications have unique process IDs encoded in the names of the files they

create so name collision can be programmatically prevented. The first component of this thesis leverages this knowledge to relax filesystem metadata synchronization and serialization so that parallel filesystem metadata accesses can be more aggressively decoupled and parallelized.

Traditionally, filesystem metadata operations are synchronously executed and serialized with respect to all concurrent filesystem metadata operations. This thesis component develops two deep filesystem metadata writeback caching protocols for relaxing this behavior. Each protocol enables a filesystem client to apply some of its filesystem metadata mutations in its private writeback cache, deferring their synchronization and serialization until an eventual bulk insertion that batch integrates all changes.

The first protocol uses locks to control and minimize potential conflicts [66]. The second protocol is more optimistic, deferring filesystem metadata synchronization and serialization without locks [92].

3.2.1 Enabling Technique: A Log-Structured Filesystem Metadata Representation

A deep client cache of filesystem metadata mutations requires an optimized on-storage filesystem metadata representation for clients to efficiently spill and access filesystem metadata information stored in an out-of-core storage. A large-scale delay in applying potentially massive amounts of filesystem metadata mutations further requires a representation that allows efficient bulk insertion.

To this end, this thesis uses a log format derived from TableFS, in which filesystem metadata is represented as a giant filesystem metadata mutation log, with each log entry having an associated row in a set of tables constructed with a modified LevelDB realization of an LSM-Tree.

Efficient filesystem namespace snapshotting. With filesystem metadata effectively logged as tables of filesystem metadata mutations, each prefix of those tables is essentially a filesystem namespace snapshot. The immutability of these tables further allows each snapshot to be shared almost trivially.

Fast metadata bulk insertion. The use of a log-structured format for filesystem metadata makes fast metadata bulk insertion possible. That is, a client wanting to insert a large amount of filesystem metadata mutations to a server can do so by constructing a table of these mutations and then directly injecting that table to the server's LSM-Tree. This allows data to be inserted to a server without being pushed through the server's write-ahead log and in-memory writeback buffers, and without the overhead of being written and formatted by the server as a logically equivalent table in the server's LSM-Tree.

Maximizing metadata mutation commutativity. To prevent metadata mutations bulk inserted by a client from conflicting with those from other clients or mutations synchronously recorded at the server, it is important for bulk inserted mutations to be as commutative as possible. Two filesystem metadata mutations are commutative if switching their application order does not change the final result. In extreme cases, mutations commutative with all other mutations can be bulk inserted into anywhere in the server's LSM-Tree without causing conflicts.

One way to maximize commutativity is to use locks and to restrict the use of bulk insertion to the creation

of new files under an empty filesystem directory tree, as the following protocol does.

3.2.2 Protocol 1: Locked Empty Trees

With filesystem metadata recorded as tables, a filesystem client may complete file creation locally if the file is known to be new and later bulk insert all file creates to the server using an efficient table insertion. This eliminates the one-RPC-per-file-create overhead seen in traditional parallel and distributed filesystems, and allows new files to be created and opened much faster, similar to the PLFS filesystem [7]. The total file creation throughput, usually dictated by the number of dedicated filesystem metadata servers a filesystem has, can be temporarily boosted as well to scale linearly with the number of clients.

Inspired by this observation, the first protocol is to enable a set of cooperative filesystem client processes to collectively lock a newly created directory. And instead of synchronously integrating filesystem metadata mutations (e.g., file creates) beneath this directory, each client process simply logs operations to be applied later in a bulk insertion.

This protocol is optimized for parallel scientific applications to decouple and parallelize their filesystem metadata activities beneath mission-critical directories (e.g., a checkpointing directory). First, being able to log and apply filesystem metadata mutations locally enables a parallel application to execute important filesystem metadata operations in parallel, and without the burden of frequent synchronization and serialization. Second, being able to bulk insert logged filesystem metadata mutations as formatted tables releases a server from having to re-execute potentially every filesystem metadata mutation performed by a client [36]. This brings down the cost of final integration.

API: a special "mkdir" operation. A filesystem client wanting to use this specific form of writeback caching issues a `mkdir` with a special flag "`LOCALIZE`", which causes a corresponding server to create the directory and return it with a renewable write lease.

During the lease period, all files and subdirectories created inside this special directory are exclusively served and recorded by the client itself. Before its lease expires, the client must return the corresponding subtree to the server, in the form of one or more tables, through the underlying storage. Once the lease expires, all bulk inserted entries will become visible to all other clients.

While the best performance is achieved when a client renews its lease many times, it may not delay bulk insertion arbitrarily. Once another client asks for access to a localized subtree, future lease renewals are denied and the lease-holding client must quickly complete its remaining bulk inserts. This prevents a client from blocking other clients indefinitely. If multiple clients want to cooperatively share a localized subtree, they can do so by each issuing a `mkdir` with a special flag "`SHARED_LOCALIZE`". In such case, all clients should complete bulk inserts before the lease expires.

Lock-controlled relaxation. With the logging and deferring of filesystem metadata changes restricted to newly created subtrees, and non-cooperative client processes locked out from seeing these subtrees, conflicts are largely avoided. There is one special case, however, for the set of client processes holding the

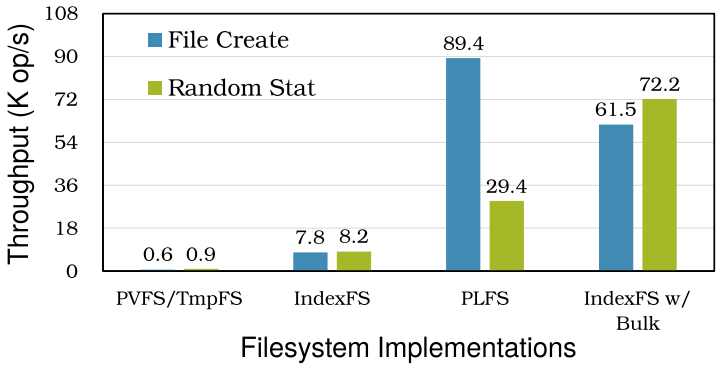


Figure 1: *Performance comparison among PVFS, PLFS, and IndexFS with and without deep client filesystem metadata writeback caching and bulk insertion. PVFS filesystem metadata was stored in an in-memory tmpfs. All runs used 1 filesystem metadata server and 16 client processes performing filesystem metadata operations. IndexFS with bulk insertion delivers comparable metadata write performance to PLFS and better metadata read throughput.*

lock to generate mutually conflicting updates. While not expected to occur frequently, conflicting updates are resolved at the server arbitrarily (e.g., by having the last to bulk insert win).

3.2.3 Protocol 1: Evaluation

Protocol 1 is prototyped as an extension to the IndexFS filesystem [66]. It is compared with PVFS [30], PLFS [7], and the original IndexFS without deep metadata writeback caching or bulk insertion.

PVFS is a traditional parallel filesystem. It manages filesystem metadata using Berkeley DB [55] and handles filesystem metadata operations synchronously using dedicated filesystem metadata servers. PLFS was originally developed to offer an N-1 interface for checkpointing while implementing only N-N checkpointing. It defers global synchronization of concurrent writes to a single file by logging the writes of each process and not serializing them until the file is read by a subsequent reader program. In addition to logging mutations of a file, PLFS techniques can also be used to log mutations of a filesystem namespace such as the creation of new files. While PLFS writes can be extremely fast, its client logging of file or namespace mutations is never understood by the underlying filesystem. Compared with PLFS, the deep client metadata writeback caching mechanism discussed in this thesis is more general purpose and more integrated with the filesystem.

Experiment results show that IndexFS with deep metadata writeback caching can create and stat files an order of magnitude faster than the IndexFS without it, which is in turn an order of magnitude faster than PVFS, as Figure 1 shows. While PLFS delivers the fastest metadata write throughput, IndexFS with deep metadata writeback caching delivers comparable write performance and better read throughput.

3.2.4 Protocol 2: Forked Namespaces ("Git" like Filesystem Metadata)

Inspired by development of the first protocol, the goal of the second protocol is to drive this deep writeback caching and bulk insertion idea further. First, instead of restricting logging and deferring to newly created subtrees, the second protocol tries to apply them to the entire namespace. Second, instead of locking out non-cooperative clients, the second protocol tries to defer synchronization and serialization without using locks. Finally, instead of focusing on decoupling and parallelizing mainly file and directory creations, the second protocol tries to allow filesystem metadata read operations to be sped up as well.

More specifically, the second protocol is to enable a set of cooperative filesystem client processes to individually checkout a specific filesystem image as a filesystem namespace snapshot. This snapshot is then used by each client process as a basis to locally complete all subsequent filesystem metadata operations until a later bulk insertion that commits all logged filesystem metadata mutations against this snapshot to the server in a single step.

Securely checking out filesystem images as snapshots. To obtain a filesystem namespace snapshot, a client issues a special "`snapshot`" call to the server. The server then flushes all its in-memory buffers to storage and returns the client with a manifest listing all tables comprising the server's current filesystem image. Since tables are designed to be immutable, they conveniently serve as a snapshot.

A filesystem client without permission to read a directory must be prohibited from reading the attributes of the files under that directory. To forbid a malicious client from accessing restricted data by scanning an entire filesystem snapshot inappropriately, filesystem servers can generate separate tables for each user-group combination such that every file referred to in an table has the same permission bits. The filesystem server then uses these permission bits to set the permission of the table in the underlying storage. This way, expected high-level access control can be correctly enforced by the underlying storage system. This mechanism is designed for environments with simple ACL practices, which we expect in most HPC platforms. Similarly, user quota control is also synchronously enforced by the underlying storage system. For ease of implementation, quota management may only apply to file data, as the size of metadata is almost always dwarfed by the size of data within an entire filesystem image.

Minimizing conflicts without locks. Unlike the first protocol in which non-cooperative clients are locked out from interfering with cooperative clients' private writeback caches, the second protocol allows concurrent modifications to a snapshotted filesystem image both from cooperative clients (through logging in private writeback caches) and non-cooperative clients (by directly updating the server's copy). To minimize potential conflicts when cooperative clients eventually flush their private writeback caches, the second protocol restricts the use of private writeback caching to only files and directories that are newly created by a client (i.e., these files and directories do not initially exist in the filesystem namespace snapshot sent by the server). Essentially, this ensures that each bulk insertion only monotonically adds information to a server, and does not modify information originally held by that server. This maximizes the commutativity of each client bulk insertion's filesystem metadata mutations, and minimizes the work a server has to do to reconcile potential conflicts.

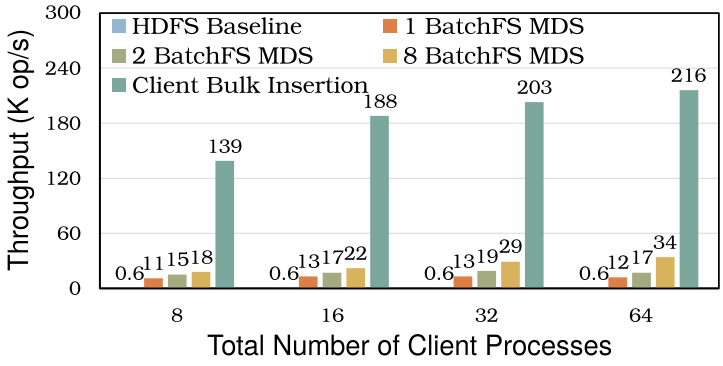


Figure 2: *Empty file creation performance of HDFS and BatchFS with and without deep client filesystem metadata writeback caching and bulk insertion. In all BatchFS runs, BatchFS used HDFS as the underlying filesystem for data and filesystem metadata storage, with 8 HDFS data servers and 1 HDFS metadata server. Different numbers of filesystem metadata servers (MDS) were deployed for BatchFS, from 1 to 8. Client caching improves performance by 8x-360x compared with directly inserting files to HDFS and to BatchFS without the client-side caching.*

3.2.5 Protocol 2: Evaluation

Protocol 2 is prototyped in BatchFS [92]. To demonstrate its potential, experiments were done using 8 NSF PRObE Kidiak machines [26] comparing BatchFS and HDFS, a widely-used open-source filesystem designed for big-data applications [73]. Each Kodiak machine has 2 CPU cores and 1 local disk.

In all experiments, a synthetic micro-benchmark tool, mdtest [41], was used to insert zero-byte files into multiple newly created directories. A total of 8-64 client processes were used. Each client process creates 1 private directory and then inserts empty files into that directory. A total 1-64 million empty files were created, depending on the total number of metadata servers each run had. All HDFS runs had 1 filesystem metadata server. Different numbers of filesystem metadata servers (MDS) were deployed for BatchFS, from 1 to 8. When client caching is enabled, each client process is counted as a filesystem metadata server. The runs were configured to generate 1 million files for each available filesystem metadata server.

Compared with synchronously serializing every filesystem metadata operation using dedicated metadata servers, allowing clients to directly apply filesystem metadata mutations leads to a more efficient filesystem metadata path and a more aggressive use of system resources. Results show that caching filesystem metadata mutations at clients can improve performance by 8x-360x compared with directly inserting files to HDFS and to BatchFS without the client-side caching, as Figure 2 shows.

3.2.6 Pending Work

Evaluation of the cost of filesystem metadata bulk insertion and subsequent server integration. It is not a surprise that client caching can improve performance at the write path. Client writeback caching is a big win when cached information can be integrated at a low price. This thesis has designed techniques for this to be the case, pending evidence proving so.

3.3 Fully-Decentralized Metadata

In addition to being developed as self-coordinated parallel batch programs, many scientific applications are configured with dedicated files for input and output, and do not necessarily benefit from a shared global filesystem namespace. The second component of this thesis is to exploit this property to further relax filesystem metadata synchronization and serialization over deep client writeback caching.

Traditionally, scientific applications co-located on a single HPC platform are forced to synchronize with a dedicated parallel filesystem namespace to obtain access to the platform's shared underlying storage. Synchronizing with a dedicated filesystem namespace can be unnecessarily taxing due to global serialization. This thesis decouples applications accessing a shared underlying storage from applications accessing a shared filesystem namespace. That is, file objects stored within a single store may be indexed by different filesystem namespaces created by different applications at different times, and a single filesystem namespace may index objects stored across different underlying stores.

Another important goal of this thesis is to decouple filesystems from dedicated services that are provisioned by, and often purchased with, the platform. To achieve this, this thesis leverages lightweight client middleware that can be dynamically instantiated by applications to provide filesystem metadata services. This empowers each application to choose from different filesystem middleware implementations, and to decide for itself the right amount of computing resources to devote to the filesystem. A metadata-intensive application may devote a large portion of its computing resources to serve filesystem metadata while a metadata-light application that reads and writes a single file may only dedicate a single CPU core to handle all filesystem metadata operations without impacting overall performance.

This thesis component develops a serverless filesystem metadata architecture for achieving these goals. It defines no global filesystem namespaces, and uses no dedicated filesystem metadata servers. Each parallel job directly instantiates a private filesystem metadata service in client middleware. These private metadata services operate on only scalable object storage, with jobs communicating with each other by sharing or publishing logs of filesystem metadata mutations. The following sections discuss the rationale behind this design, and outline its key components.

3.3.1 Rationale behind No Ground Truth

The first step to avoid global synchronization is to have applications individually define their own filesystem namespaces. To reason about the impact of an absence of a global filesystem namespace, however, this thesis considers the following cases.

No data sharing happens when a group of applications access mutually disjoint sets of data. Having no global filesystem namespace won't be a problem when there is no sharing.

Sequential data sharing happens when the output of an application is subsequently used as input for another application. Removing a global filesystem won't be a problem for sequentially sharing applications as long as there is a mechanism for data to be propagated from the writing application to the later

reading application.

Concurrent data sharing happens when multiple related applications write data into a single set of files without pre-deciding write order. In the absence of a global filesystem namespace, one of these applications can be elected as a master application, with its namespace then shared among all other follower applications either in the conventional synchronous manner or using a mechanism previously discussed (§3.2) to improve parallelism.

Anonymous synchronization happens when a group of applications use the namespace of a filesystem primarily to achieve synchronization. For example, two application processes may use a "`.LOCK`" file to achieve mutual exclusiveness [44].

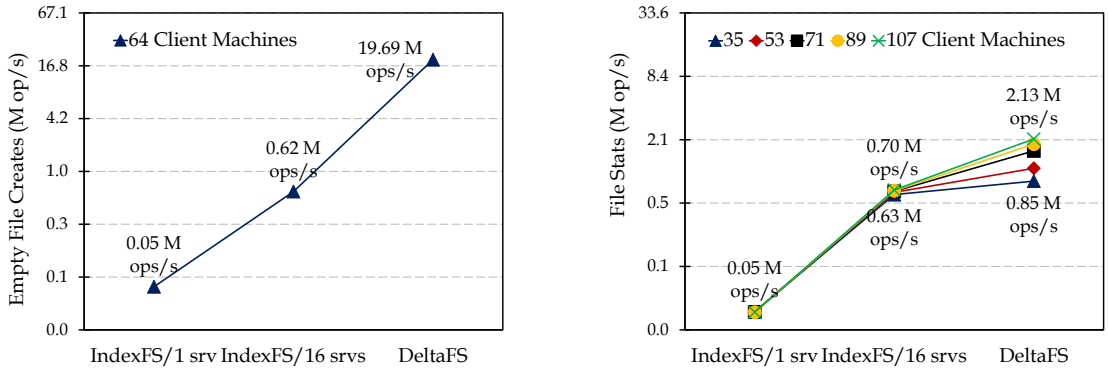
Abstractly, filesystems were originally developed for applications to more easily access data stored in an underlying storage. But with filesystem namespaces typically implemented with immediate global visibility using transactional properties such as atomic renames and instant name collision checks, filesystems have been sometimes used by applications to achieve communication beyond single data sharing. By removing a global filesystem namespace, this thesis aims to restore filesystems to their original use-cases. In the absence of a global filesystem namespace as a synchronization service, applications should seek anonymous synchronization through a purpose-built mechanism outside of the filesystem, such as a dedicated lock service [11, 32], or a shared message bus.

3.3.2 A Serverless Filesystem Metadata Architecture

This thesis envisions a serverless filesystem metadata architecture consisting of the following components.

Client filesystem metadata middleware is filesystem library code linked at each application's process. It is instantiated by each application process during bootstrapping and it provides each application process with a private filesystem namespace that is separated from other filesystem namespaces. Each such middleware instance can be viewed as a full-fledged but embedded filesystem metadata server capable of executing filesystem metadata operations and writing logs of metadata mutations. Each application process takes one or more filesystem namespace snapshots (logs of filesystem metadata mutations) as input, and publishes its output as a new snapshot ready to be consumed by subsequent application processes.

Namespace snapshot registries. Instead of synchronizing with a global filesystem namespace for sequential data sharing, unrelated applications communicate with each other through sharing and publishing filesystem namespace snapshots. Each filesystem namespace snapshot is essentially a filesystem metadata mutation log, recording filesystem metadata mutations relative to one or more base filesystem namespace snapshots. In lieu of a "global filesystem namespace", one or more external namespace registries can be deployed to serve as repositories of published filesystem namespace snapshots. The insertion, deletion, and selection of snapshots from these registries captures the true communication and synchronization between unrelated applications. It is also possible for an integrated workflow engine to automate namespace propagation such that a follow-up workflow step directly inherits namespaces from its parent steps rather than looking them up externally through a public registry.



(a) *Empty file creation throughput of DeltaFS and different IndexFS configurations*

(b) *File open throughput of DeltaFS and different IndexFS configurations*

Figure 3: *Performance of creating and stating empty files within a single directory using IndexFS or DeltaFS on 124 NFS PRObE Nome machines. The experiments compared running IndexFS on 1 and 16 dedicated server machines, and running DeltaFS as client middleware on up to 107 client machines.*

Snapshot search engines. With snapshot registries, filesystem namespace snapshots can be searched by their names. To make snapshots searchable by other attributes (e.g., pathnames of files within a snapshot), one or more snapshot search engines can be deployed to more thoroughly index snapshots. Each search engine subscribes to one or more snapshot registries, and periodically updates its indexes.

3.3.3 Preliminary Results

The idea of a serverless filesystem is prototyped in DeltaFS. Preliminary experiments were done comparing DeltaFS and IndexFS on 124 NSF PRObE Nome machines [26]. Each Nome machine has 16 CPU cores and 32GB of memory. The experiments compared running IndexFS using 1 and 16 dedicated server machines, and running DeltaFS as client middleware on up to 107 client machines.

Figure 3a shows the performance of empty file creates under a shared directory. All runs used 512 synthetic client processes on 64 client machines. DeltaFS delivers up to 3 orders of magnitude more throughput thanks to a more efficient metadata path that avoids the use of RPC and being able to utilize client resources to process filesystem metadata operations.

Figure 3b compares the performance of random file stats using different numbers of client machines, from 35 to 107. Note that these numbers were deliberately selected to be poorly aligned with each other, emphasizing the fact that DeltaFS is able to scale flexibly, without requiring the read phase job to be perfectly aligned with the job at the write phase.

The throughput for both IndexFS runs increases as the total amount of dedicated server resources increase. Not having any dedicated metadata servers, DeltaFS jobs each use the embedded DeltaFS code to perform filesystem metadata operations. As a larger set of machines were used to serve filesystem meta-

data, DeltaFS jobs are able to achieve higher metadata read throughput. The throughput increases as job scales.

3.4 Embedded In-Situ Data Processing

At exascale, scientific simulations will reach unprecedented levels of scale and complexity. The resulting datasets require efficient data pipelines to quickly unlock the scientific insight buried within the data. The last component of this thesis presents an embedded in-situ data processing mechanism dynamically transforming write-optimized data to a read-optimized format without incurring significant extra storage I/O activities. To achieve this, a simulation's data output is on the fly partitioned among the simulation's job processes and indexed with a modified LSM-Tree at each data partition. To minimize extra storage I/O, this thesis partitions and indexes data as, *and only as*, data is written to storage the very first time (i.e., write-once, no reading back or re-writing). Data partitioning maps each query to a single partition, while the indexes dynamically created at each partition further accelerate per-partition data lookups.

3.4.1 Design Goals

A) Fast queries without requiring O(data size) post-processing. Consider a scientific simulation that runs in timesteps. To trace its progress, simulation state is periodically persisted to storage for post-analysis. Traditionally, efficient analysis of large-scale simulation output can be achieved through careful data post-processing that may be prohibitively time-consuming. This thesis processes data in-situ, with a goal of making post-processing unnecessary for certain classes of post-analysis queries. Designing this in-situ process as an embedded function further prevents it from requiring dedicated computing resources.

B) Resource-obliviousness. For experiments that are allocated with limited computing hours, simulation output is not necessarily analyzed on the main computing cluster, and analysis may be performed on a side cluster. Compared to the main cluster, side clusters possess less computational power and enjoy less total bandwidth to the underlying storage. Another important goal of this thesis component is to enable resource-oblivious scientific analysis such that data can be quickly queried even using a small side cluster, and in extreme cases, with perhaps a laptop that is equipped with only a single CPU core.

3.4.2 Key Idea: Reusing Idle CPU Cycles to Perform In-situ Data Reorganization and Indexing

For simulations whose state is written to storage synchronously (i.e., the simulation does not overlap storage I/O with simulation computation), the writing process is expected to be limited by and blocked on storage. This leaves idle CPU cycles on compute nodes of the simulation and this thesis reuses these idle CPU cycles to perform in-situ data operations. In addition, because the simulation is effectively forced to pause its computation during its I/O phases, the compute nodes' interconnection network, often faster than the storage system, is leveraged by this thesis as well to serve in-situ activities.

3.4.3 Targeted Workloads and Data Modeling

Highly-selective post-analysis queries. This thesis component targets simulations whose post-analysis consists of queries that are highly-selective, and knowledge about the data to be queried is not known until the simulation concludes. That is, a scientist does not necessarily know what to query before the simulation ends and may rely on the final state of the simulation to determine what is interesting. Because all keys may be queried, instead of being a potential waste of work, it is important for all data to be in-situ indexed such that queries against any specific key can be answered efficiently.

This thesis models data as simple key-value pairs. Queries are performed only through keys. Each query targets one specific key and returns data for it across a series of timesteps. This thesis targets large-scale simulations with trillions of keys generated per timestep and hundreds of timesteps per simulation.

3.4.4 Embedded Data Partitioning

To partition data, an application's data output is hashed among the application's processes via the platform's interconnection network. While data partitioning is important in constructing an optimized storage layout for efficient data analysis, performing it within an embedded function across a large number of processes and under intensive memory pressure is non-trivial.

Aggressive writeback buffering. Even with today's fast HPC interconnection network, this thesis has found it imperative to aggressively buffer data before sending a batch of it over to the network so communication costs can be adequately hidden and amortized. For situations where all processes send data to all other processes, every process needs to buffer data to be sent to every other process. Unfortunately, for large-scale simulations with hundreds of thousands of processes, the total amount of memory required for efficient use of the network through buffering at each process will quickly become unacceptable to application programmers sharing the same memory, making direct all-to-all communication (i.e., each process directly sends messages to each other process) infeasible.

Scalable all-to-all communication through multi-hop routing. To restrict memory use in large-scale application runs, this thesis routes messages via multiple hops with each application process potentially simultaneously acting as a sender, a receiver, and an intermediate message forwarder. Messages are no longer directly sent to their final destinations. Instead, application processes each forward some of their messages to other processes in the application to progress message delivery such that each process only needs to communicate with a small subset of its peers. This prevents a process from having to make and maintain connections to all other processes, and better limits the per-process memory state.

Avoiding overloading the network. Even with fast HPC interconnects, all-to-all data movement across a large number of application processes and compute nodes can be prohibitively expensive. This is especially true when the compute nodes performing the network communication is equipped with slow-spinning processors [3, 42, 52, 74] for cost-effective high computing throughput (i.e., total flops per machine) such that single-threaded event handlers executed at critical regions (e.g., performing software tag matching at the NICs) are no longer fast enough to meet latency targets [51].

To reduce shuffle overhead, this thesis uses a scheme that decreases the total amount of messages sent through the network when performing online data partitioning. The key idea is to persist each key-value pair to local or shared storage directly, moving it quickly off the network. Then partitioning is performed on a compact representation of the key-value pairs. This compact representation consists of a prefix derived from the key and the ID of the process that generated the key-value pair. This representation is more compact than previous work that moves keys and pointers to data [49, 66].

3.4.5 Embedded Data Indexing

Traditionally, efficient read and write performance is achieved by allocating a small in-memory data structure (e.g., a B-Tree) to absorb incoming writes and then merging the data into a larger read-optimized on-disk data structure (e.g., a larger B-Tree) whenever the small in-memory data structure is full. This log-structured merge-mechanism forms the basis of many high-performance data systems today [28, 38, 45, 80, 82]. But the need to repeatedly read back data from storage and re-merge it with incoming data renders this mechanism prohibitively expensive. This is particularly true when the process is running within an embedded in-situ data function co-located with a scientific application, in which sufficient runnable windows to complete even the lightest form of log-structured merging, or any other forms of post-write data reorganization, may not be available on compute nodes. This exemplifies a key difference between an embedded I/O service and a dedicated data server: a dedicated server is able to leverage ownership of resources to schedule a significant portion of its work in the background whereas to minimize interference, an embedded data function must co-schedule all its work with application I/O.

Indexing data without post-write data reorganization. With queries restricted to individual data partition, optimizing the per-partition storage layout can further speed up query processing. When dedicated resources are available, an indexing program could, at the cost of incurring a potentially large number of storage reads and re-writes, keep integrating incoming data into an on-disk data structure such that the total number of data locations a query needs to check for any specific key is always logarithmically bound to the total size of data in a data partition [34, 56]. This frees queries from having to perform linear searches and enables each of them to check data only at a small number of places.

To speed up reads without requiring costly post-write data reorganization operations, however, this thesis logs data as a series of data subsequences and for each constructs a filter able to test whether a key *might*, or *must not*, exist in the filter's paired data subsequence [9, 23]. This enables subsequent queries to each skip data subsequences that are known to be irrelevant and not spend time reading their data, thus allowing data to be recalled with fewer storage seeks.

Clustered indexes. Scientific analysis queries typically start with a code cache, and may not possess enough locality to benefit from caching generally. To retrieve data from storage, the indexes (and the filters) of the data are often needed and must be read into memory before data can be read. Reading data indexes can be costly when index data is spread over multiple non-continuous locations in storage and is not cached in memory. To alleviate this overhead, this thesis packs index data based on expected query types allowing storage bandwidth to be more fully utilized during analysis queries [38, 85].

Parallel reads. To further speed up queries, the underlying storage system's parallelism can be leveraged to search multiple data locations in parallel. This allows data to be found faster [49].

3.4.6 Example Application: VPIC

To evaluate and better explain in-situ data processing, this thesis uses LANL's VPIC (Vector Particle-In-Cell) code [10] as an example.

Time-base simulations. VPIC applications are time-based particle simulations. In a VPIC simulation, each simulation process manages a region of cells in the simulation space through which particles move. Every few timesteps the simulation stops and each simulation process writes a per-process file containing the state of all the particles currently managed by the process. State for each particle is 48 bytes. Large-scale VPIC simulations have been conducted with trillions of particles, generating terabytes of data for each recorded timestep [12, 13].

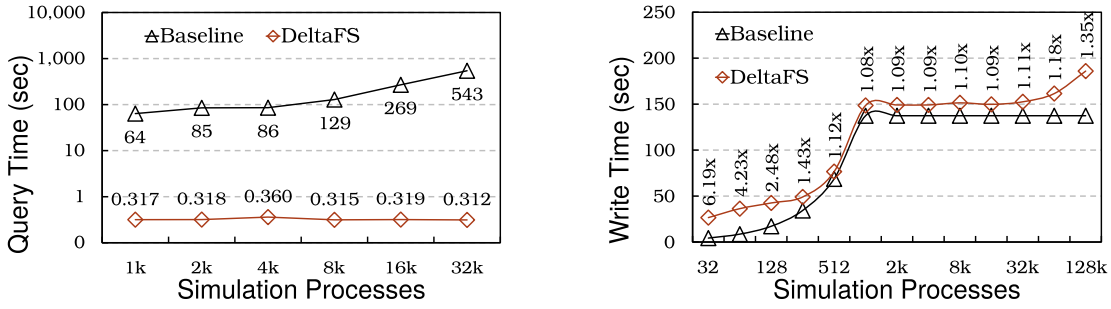
A needle-in-a-haystack problem. VPIC domain scientists are often interested in the behavior of a tiny subset of particles with specific characteristics (e.g., high energy). Unfortunately, because particles can move drastically within the simulation space during a simulation, information about a specific particle is typically recorded in different per-process output files at different timesteps. Consequently, even though the IDs of interested particles can be known at the end of a simulation, searching and reading back the trajectories of these particles is like finding a needle in a haystack. That is, without data reorganization a complete scan of the entire simulation output may be needed to recall per-particle information.

To speed up VPIC queries, this thesis applies in-situ processing to VPIC's particle output: dynamically partitioning and indexing particle data as it is written to storage can enable per-particle information to be efficiently recalled afterwards without requiring reading back potentially an entire particle dataset.

3.4.7 Realization: Embedded In-Situ Data Processing as a Special Directory Type

For ease of application adoption, this thesis has chosen to realize in-situ data processing as a software-defined filesystem directory type, referred to as the *Indexed Massive Directory*, that can be efficiently instantiated on client middleware [90]. Traditionally, simulation output is directly streamed to per-process output files (i.e., N-N writing). To have data in-situ reorganized for efficient reads during the writing of the data, one programs their simulations to write data into an Indexed Massive Directory.

The Indexed Massive Directory API. Indexed Massive Directories are a special type of directory that are designed to work like a regular directory. While at their simplest, these directories typically contain a massive number of tiny files. Underneath each such directory is an embedded in-situ data processing pipeline running inside the distributed processes of a parallel simulation. The purpose of having a directory-oriented interface is to bridge the gap between scientific applications and the underlying data pipeline: scientific applications expect files whereas the underlying data pipeline speaks in keys and values. To fill this gap, file data written to an Indexed Massive Directory is transformed to key-value pairs



(a) *Performance of particle trajectory queries under a baseline reader and DeltaFS*

(b) *Performance of simulation I/O and overhead of DeltaFS in-situ processing*

Figure 4: *Results from real VPIC simulation runs on LANL's Trinity hardware. The biggest run used 4,096 compute nodes, 131,072 CPU cores, simulated 2 trillion particles, and generated approximately 480TB of data. The baseline reader program used all the CPU cores to search particles in parallel, whereas all DeltaFS queries were executed on a single CPU core.*

and sent to the underlying data pipeline: filenames become keys and file contents become opaque values. Values are appended to keys as data is appended to files. As each directory is backed by an embedded in-situ data processing pipeline, file data written to these directories is dynamically partitioned and indexed, with both keyed on filenames. Partitioned and indexed file data is packed and stored as per-partition log files in the underlying storage, similar to the PLFS filesystem [7]. After a simulation, files can be efficiently recalled from a massive directory using filenames as the keys.

Case study: one file for each particle. Consider a VPIC simulation. Interested in the trajectories of a tiny subset of particles whose identities are not known until the end of a simulation, VPIC could benefit from a dynamically constructed lookup table mapping particle IDs to particle trajectories so that any particle's trajectory can be efficiently recalled following each simulation. To achieve this, VPIC instantiates an Indexed Massive Directory for each simulation and creates a file for each particle it simulates. Each file is named by the ID of the particle it represents. During the course of a simulation, particle state at different timesteps is appended to the same per-particle files. Since every Indexed Massive Directory is constructed to dynamically reorganize itself for its files to be efficiently recalled and each file is written with the entire trajectory of a specific particle, it directly serves as a particle trajectory lookup table for VPIC at the end of a simulation. To retrieve the trajectory of a specific particle, a reader program uses the Indexed Massive Directory API to open and access the corresponding file of that particle. Internally, the Indexed Massive Directory uses its per-partition log files to quickly locate the data of that file. This data location process is transparent to the reader program. Located file data, which is opaque to the Indexed Massive Directory, is read by the reader program and interpreted by it as a particle trajectory.

3.4.8 Demonstration and Results

To demonstrate the potential of in-situ data processing, experiments were done using a prototype implementation of the Indexed Massive Directory, DeltaFS, and the VPIC application (introduced in §3.4.6) on

the LANL's Trinity computing cluster. A maximum of 4,096 Trinity compute nodes were used, amounting to 131,072 CPU cores. Figure 4 compares the performance of running VPIC simulations and performing VPIC particle trajectory queries with and without DeltaFS in-situ processing.

Experiment setup. Each experiment consisted of a real VPIC simulation configuration both with and without DeltaFS. Each simulation run had one simulation process on each CPU core. For VPIC baseline runs, the simulation wrote one output file per simulation process. For DeltaFS runs, the VPIC simulation wrote into an Indexed Massive Directory, with DeltaFS dynamically partitioning and indexing the data, and writing the results as parallel logs. Both the data partitioning and the indexing were keyed on particle IDs, and the data was partitioned using a hash function. The largest run simulated 2 trillion particles across 131,072 simulation processes.

Across all runs, simulation data was first written to a fast burst-buffer storage tier and was later staged out to an underlying Lustre file system. The experiments kept the compute node to burst-buffer node ratio fixed at 32 to 1. Writing data from compute nodes to burst-buffer nodes is bottlenecked on the burst-buffer node's NIC's. Each burst-buffer node can absorb data at approximately 5.3GB per second.

After each simulation, queries were executed directly from the underlying file system with each query targeting a random particle and reading all of its data. Particle data was written out over time as the simulation ran through timesteps. Each simulation was configured to output all particle data for 5 of those timesteps. Each query therefore returns data from 5 distinct points in time. To retrieve the trajectory of a particle, the VPIC baseline reader always reads the entire simulation output and each query was repeated only 1 or 2 times. As DeltaFS can handle queries more efficiently, all DeltaFS queries were repeated 100 times. Each query started with a cold data cache. The average query latency is reported. DeltaFS used a single CPU core to execute queries, whereas the baseline reader used the number of simulation processes to read data in parallel.

Orders of magnitude faster queries. Figure 4a shows the read performance. While the baseline reader used all the CPU cores to run queries, a single-core DeltaFS reader was still up-to 1,740x faster. This is because without an index for particles, the baseline reader reads all the particle data so its query latency is largely bounded by the underlying storage bandwidth. As DeltaFS builds indexes in-situ, it is able to quickly locate per-particle information after a simulation and maintain a low query latency (about 300ms in these experiments) as the simulation scales.

Small write overhead. Figure 4b shows the I/O overhead DeltaFS adds to the simulation's I/O phases for building the data indexes. Part of the overhead comes from writing the indexes in addition to the original simulation output. The rest is due to the reduced I/O efficiency resulting from DeltaFS performing the in-situ indexing work. DeltaFS had large but decreasing overheads for the first 5 runs. This is because those jobs are not large enough to saturate the burst-buffer storage, so the system is dominated by the extra work DeltaFS performs to build the indexes. Starting from the sixth run the jobs began to bottleneck on the storage, and there is a modest DeltaFS slowdown of about 10%. For the last 2 runs, the job sizes are deliberately increased to demonstrate the performance at scale, and there is a slowdown of 20%-35%.

4 Conclusion

Every once in a while, a revolutionary paradigm shift comes along that dramatically changes the way people build filesystems. One such breakthrough was made by the NASD project, which asked people to stop serving data from the filesystem's metadata plane [25]. For decades, this idea has been the underpinning of enabling scalable bandwidth from the object storage systems that underlie modern distributed and parallel filesystems [15, 24, 31, 72, 81, 84]. This thesis informs people of what it takes to further adapt filesystems to future HPC platforms. With the convergence of HPC and big data, the changes put forth in this thesis may prove necessary for other areas of computing as well.

At exascale and beyond, synchronization of anything global should be avoided, even if the changes needed to do so are drastic. Conventional parallel filesystems, with fully synchronous namespaces, mandate synchronization with file create and other filesystem metadata operations. This must stop. Moreover, the idea of dedicating a single filesystem metadata service to meet the needs of all applications running on a single computing environment, is archaic and inflexible. This too must stop. By shifting away from global namespaces and transforming dedicated filesystem metadata servers to per-job client software, this thesis breaks established filesystem designs, and identifies the changes needed for future scalable file system metadata. Furthermore, by designing and implementing Indexed Massive Directories, this thesis demonstrates the opportunity for the use of storage writeback buffering to serve in-situ operations, and shows an effective mechanism for efficiently achieving this, even at scale.

References

[1] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni. "Parallel I/O and the Metadata Wall." In: *Proceedings of the Sixth Workshop on Parallel Data Storage (PDSW 11)*. 2011, pp. 13–18. DOI: 10.1145/2159352.2159356 (cit. on p. 1).

[2] *ANL Aurora*. https://www.alcf.anl.gov/alcf-aurora-2021-early-science-program-data-and-learning-call-proposals. 2018 (cit. on p. 1).

[3] *ANL Theta*. https://www.alcf.anl.gov/theta/ (cit. on p. 20).

[4] J. C. Bennett, H. Abbasi, P. T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen. "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis." In: *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 12)*. 2012, pp. 1–9. DOI: 10.1109/SC.2012.31 (cit. on p. 9).

[5] 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: *Proceedings of the 2012 IEEE Conference on Massive Storage Systems and Technologies (MSST 12)*. 2012, pp. 1–5. DOI: 10.1109/MSST.2012.6232382 (cit. on p. 9).

[6] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez. "Storage challenges at Los Alamos National Lab." In: *Proceedings of the 2012 IEEE Conference on Massive Storage Systems and Technologies (MSST 12)*. 2012, pp. 1–5. DOI: 10.1109/MSST.2012.6232376 (cit. on pp. 5, 8).

[7] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. "PLFS: A Checkpoint Filesystem for Parallel Applications." In: *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 09)*. 2009, 21:1–21:12. DOI: 10.1145/1654059.1654081 (cit. on pp. 2, 5, 9, 10, 12, 13, 23).

[8] J. Bent, B. Settlemyer, and G. Grider. "Serving Data to the Lunatic Fringe: The Evolution of HPC Storage." In: *USENIX ;login:* 41.2 (June 2016) (cit. on pp. 5, 8, 9).

[9] B. H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors." In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. DOI: 10.1145/362686.362692 (cit. on p. 21).

[10] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. "Ultrahigh Performance Three-dimensional Electromagnetic Relativistic Kinetic Plasma Simulation." In: *Physics of Plasmas* 15.5 (2008), p. 7 (cit. on p. 22).

[11] M. Burrows. "The Chubby Lock Service for Loosely-coupled Distributed Systems." In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*. 2006, pp. 335–350 (cit. on p. 17).

[12] S. Byna, R. Sisneros, K. Chadalavada, and Q. Koziol. "Tuning Parallel I/O on Blue Waters for Writing 10 Trillion Particles." In: *Cray User Group (CUG)*. https://cug.org/proceedings/cug2015_proceedings/includes/files/pap120-file2.pdf. 2015 (cit. on p. 22).

[13] S. Byna, A. Uselton, D. K. Prabhat, and Y. He. "Trillion particles, 120,000 cores, and 350 TBs: Lessons learned from a hero I/O run on Hopper." In: *Cray User Group (CUG)*. https://cug.org/proceedings/cug2013_proceedings/includes/files/pap107-file2.pdf. 2013 (cit. on p. 22).

[14] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughton, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu. "Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation." In: *Proceedings of the 2012 International Conference on High Performance Computing, Networking, Storage, and Analysis (SC 12)*. 2012, 59:1–59:12. DOI: 10.1109/SC.2012.92 (cit. on p. 1).

[15] P. Carns, W. Ligon, R. Ross, and R. Thakur. "PVFS: A Parallel File System for Linux Clusters." In: *Proceedings of the 4th Annual Linux Showcase & Conference (ALS 00)*. 2000, pp. 317–328 (cit. on pp. 1, 25).

[16] K. Chasapis, M. F. Dolz, M. Kuhn, and T. Ludwig. "Evaluating Lustre's Metadata Server on a Multi-socket Platform." In: *Proceedings of the 9th Parallel Data Storage Workshop (PDSW 14)*. 2014, pp. 13–18. DOI: 10.1109/PDSW.2014.5 (cit. on p. 1).

[17] J. Chou, K. Wu, and Prabhat. "FastQuery: A Parallel Indexing System for Scientific Data." In: *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER 11)*. 2011, pp. 455–464. DOI: 10.1109/CLUSTER.2011.86 (cit. on pp. 2, 8).

[18] J. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E. W. Bethel, A. Shoshani, O. Rübel, Prabhat, and R. D. Ryne. "Parallel Index and Query for Large Scale Data Analysis." In: *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 11)*. 2011, 30:1–30:11. DOI: 10.1145/2063384.2063424 (cit. on p. 8).

[19] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *Proceedings of the 6th Symposium on Opearting Systems Design and Implementation (OSDI 04)*. 2004, pp. 10–10 (cit. on p. 1).

[20] B. Dong, S. Byna, and K. Wu. "Expediting scientific data analysis with reorganization of data." In: *Proceedings of the 2013 IEEE International Conference on Cluster Computing (CLUSTER 13)*. 2013, pp. 1–8. DOI: 10.1109/CLUSTER.2013.6702675 (cit. on pp. 2, 8).

[21] J. R. Douceur and J. Howell. "Distributed Directory Service in the Farsite File System." In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*. 2006, pp. 321–334 (cit. on pp. 5, 7).

[22] *Exascale Computing Project*. https://www.exascaleproject.org/. 2018 (cit. on p. 1).

[23] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. "Cuckoo Filter: Practically Better Than Bloom." In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT 14)*. 2014, pp. 75–88. DOI: 10.1145/2674005.2674994 (cit. on p. 21).

[24] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google File System." In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP 03)*. 2003, pp. 29–43. DOI: 10.1145/945445.945450 (cit. on pp. 1, 25).

[25] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. "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)*. 1998, pp. 92–103. DOI: 10.1145/291069.291029 (cit. on pp. 1, 25).

[26] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. "PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research." In: *USENIX ;login:* 38.3 (June 2013) (cit. on pp. 15, 18).

[27] D. K. Gifford, R. M. Needham, and M. D. Schroeder. "The Cedar File System." In: *Commun. ACM* 31.3 (Mar. 1988), pp. 288–298. DOI: 10.1145/42392.42398 (cit. on p. 1).

[28] H. N. Greenberg, J. Bent, and G. Grider. "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)*. 2015, pp. 10–10 (cit. on p. 21).

[29] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari. "Failures in Large Scale Systems: Long-term Measurement, Analysis, and Implications." In: *Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 17)*. 2017, 44:1–44:12. DOI: 10.1145/3126908.3126937 (cit. on p. 5).

[30] I. F. Haddad. "PVFS: A Parallel Virtual File System for Linux Clusters." In: *Linux J.* 2000.80es (Nov. 2000) (cit. on p. 13).

[31] D. Hildebrand and P. Honeyman. "Exporting Storage Systems in a Scalable Manner with pNFS." In: *Proceedings of the 2005 IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 05)*. 2005, pp. 18–27. DOI: 10.1109/MSST.2005.14 (cit. on pp. 1, 25).

[32] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC 10)*. 2010, pp. 11–11 (cit. on p. 17).

[33] J. Inman, W. Vining, G. Ransom, and G. Grider. "MarFS, a Near-POSIX Interface to Cloud Objects." In: *USENIX ;login:* 42.1 (Jan. 2017) (cit. on p. 9).

[34] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. "Incremental Organization for Data Recording and Warehousing." In: *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB 97)*. 1997, pp. 16–25 (cit. on p. 21).

[35] D. P. Jasper. "A discussion of checkpoint/restart." In: *Software Age* (Oct. 1969) (cit. on p. 5).

[36] J. J. Kistler and M. Satyanarayanan. "Disconnected Operation in the Coda File System." In: *ACM Trans. Comput. Syst.* 10.1 (Feb. 1992), pp. 3–25. DOI: 10.1145/146941.146942 (cit. on p. 12).

[37] H. T. Kung and J. T. Robinson. "On Optimistic Methods for Concurrency Control." In: *ACM Trans. Database Syst.* 6.2 (June 1981), pp. 213–226. DOI: 10.1145/319566.319567 (cit. on p. 7).

[38] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System." In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. DOI: 10.1145/1773912.1773922 (cit. on p. 21).

[39] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. "I/O Performance Challenges at Leadership Scale." In: *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 09)*. 2009, 40:1–40:12. DOI: 10.1145/1654059.1654100 (cit. on p. 8).

[40] *LANL Crossroads*. https://www.lanl.gov/projects/crossroads/. 2018 (cit. on p. 1).

[41] *LANL mdtest*. https://github.com/MDTEST-LANL/mdtest. 2017 (cit. on p. 15).

[42] *LANL Trinity*. http://www.lanl.gov/projects/trinity/. 2018 (cit. on p. 20).

[43] LANL, NERSC, and SNL. *APEX Workflows*. https://www.nersc.gov/assets/apex-workflows-v2.pdf. Mar. 2016 (cit. on p. 3).

[44] *LevelDB*. https://github.com/google/leveldb/ (cit. on p. 17).

[45] S. Li, Y. Lu, J. Shu, Y. Hu, and T. Li. "LocoFS: A Loosely-coupled Metadata Service for Distributed File Systems." In: *Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 17)*. 2017, 4:1–4:12. DOI: 10.1145/3126908.3126928 (cit. on p. 21).

[46] 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: *Proceedings of the 2012 IEEE Conference on Massive Storage Systems and Technologies (MSST 12)*. 2012, pp. 1–11. DOI: 10.1109/MSST.2012.6232369 (cit. on p. 8).

[47] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu. "Six Degrees of Scientific Data: Reading Patterns for Extreme Scale Science IO." In: *Proceedings of the 20th International Symposium on High Performance Distributed Computing (HPDC 11)*. 2011, pp. 49–60. DOI: 10.1145/1996130.1996139 (cit. on p. 2).

[48] J. Lofstead and R. Ross. "Insights for Exascale IO APIs from Building a Petascale IO API." In: *Proceedings of the 2013 International Conference on High Performance Computing, Networking, Storage, and Analysis (SC 13)*. 2013, 87:1–87:12. DOI: 10.1145/2503210.2503238 (cit. on p. 8).

[49] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "WiscKey: Separating Keys from Values in SSD-conscious Storage." In: *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST 16)*. 2016, pp. 133–148 (cit. on pp. 21, 22).

[50] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. "Andrew: A Distributed Personal Computing Environment." In: *Commun. ACM* 29.3 (Mar. 1986), pp. 184–201. DOI: 10.1145/5666.5671 (cit. on p. 1).

[51] T. Mudge and U. Holzle. "Challenges and Opportunities for Extremely Energy-Efficient Processors." In: *IEEE Micro* 30.4 (July 2010), pp. 20–24. DOI: 10.1109/MM.2010.61 (cit. on p. 20).

[52] *NERSC Cori*. https://www.nersc.gov/users/computational-systems/cori/ (cit. on p. 20).

[53] A. Nisar, W. k. Liao, and A. Choudhary. "Scaling parallel I/O performance through I/O delegate and caching system." In: *Proceedings of the 2008 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 08)*. 2008, pp. 1–12. DOI: 10.1109/SC.2008.5214358 (cit. on p. 9).

[54] R. A. Oldfield, G. D. Sjaardema, G. F. Lofstead II, and T. Kordenbrock. "Trilinos I/O Support Trios." In: *Sci. Program.* 20.2 (Apr. 2012), pp. 181–196. DOI: 10.1155/2012/842791 (cit. on p. 9).

[55] M. A. Olson, K. Bostic, and M. Seltzer. "Berkeley DB." In: *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX ATC 99)*. 1999, pp. 43–43 (cit. on p. 13).

[56] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. "The Log-structured Merge-tree (LSM-tree)." In: *Acta Inf.* 33.4 (June 1996), pp. 351–385. DOI: 10.1007/s002360050048 (cit. on pp. 2, 5, 21).

[57] S. Oral, J. Simmons, J. Hill, D. Leverman, F. Wang, M. Ezell, R. Miller, D. Fuller, R. Gunasekaran, Y. Kim, S. Gupta, D. Tiwari, S. S. Vazhkudai, J. H. Rogers, D. Dillow, G. M. Shipman, and A. S. Bland. "Best Practices and Lessons Learned from Deploying and Operating Large-scale Data-centric Parallel File Systems." In: *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 14)*. 2014, pp. 217–228. DOI: 10.1109/SC.2014.23 (cit. on p. 1).

[58] *ORNL Summit*. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/. 2018 (cit. on p. 1).

[59] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. "The Sprite network operating system." In: *Computer* 21.2 (Feb. 1988), pp. 23–36. DOI: 10.1109/2.16 (cit. on pp. 1, 7).

[60] S. Patil and G. Gibson. "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)*. 2011, pp. 13–13 (cit. on p. 5).

[61] M. Polte, J. Simsa, W. Tantisiriroj, G. Gibson, S. Dayal, M. Chainani, and D. K. Uppugandla. "Fast log-based concurrent writing of checkpoints." In: *Proceedings of the 3rd Annual Workshop on Petascale Data Storage (PDSW 08)*. Nov. 2008, pp. 1–4. DOI: 10.1109/PDSW.2008.4811882 (cit. on p. 5).

[62] M. Polte, J. Lofstead, J. Bent, G. Gibson, S. A. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, M. Wingate, and M. Wolf. "...And Eat It Too: High Read Performance in Write-optimized HPC I/O Middleware File Formats." In: *Proceedings of the 4th Annual Workshop on Petascale Data Storage (PDSW 09)*. 2009, pp. 21–25. DOI: 10.1145/1713072.1713079 (cit. on p. 5).

[63] R. Rajachandrasekar, A. Moody, K. Mohror, and D. K. Panda. "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)*. 2013, pp. 143–154. DOI: 10.1145/2462902.2462908 (cit. on p. 9).

[64] K. Ren and G. Gibson. "TABLEFS: Enhancing Metadata Efficiency in the Local File System." In: *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 2013, pp. 145–156 (cit. on pp. 1, 2, 5).

[65] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. "SlimDB: A Space-efficient Key-value Storage Engine for Semi-sorted Data." In: *Proc. VLDB Endow.* 10.13 (Sept. 2017), pp. 2037–2048. DOI: 10.14778/3151106.3151108 (cit. on p. 5).

[66] K. Ren, Q. Zheng, S. Patil, and G. Gibson. "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)*. 2014, pp. 237–248. DOI: 10.1109/SC.2014.25 (cit. on pp. 1, 2, 6, 11, 13, 21).

[67] M. Rosenblum and J. K. Ousterhout. "The Design and Implementation of a Log-structured File System." In: *ACM Trans. Comput. Syst.* 10.1 (Feb. 1992), pp. 26–52. DOI: 10.1145/146941.146943 (cit. on p. 8).

[68] N. El-Sayed and B. Schroeder. "Checkpoint/restart in practice: When 'simple is better'." In: *Proceedings of the 2014 IEEE International Conference on Cluster Computing (CLUSTER 14)*. 2014, pp. 84–92. DOI: 10.1109/CLUSTER.2014.6968777 (cit. on p. 5).

[69] N. El-Sayed and B. Schroeder. "To checkpoint or not to checkpoint: Understanding energy-performance-I/O tradeoffs in HPC checkpointing." In: *Proceedings of the 2014 IEEE International Conference on Cluster Computing (CLUSTER 14)*. 2014, pp. 93–102. DOI: 10.1109/CLUSTER.2014.6968778 (cit. on p. 5).

[70] F. B. Schmuck and R. L. Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters." In: *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*. 2002, pp. 231–244 (cit. on pp. 1, 5, 7).

[71] B. Schroeder and G. Gibson. "A Large-Scale Study of Failures in High-Performance Computing Systems." In: *IEEE Transactions on Dependable and Secure Computing* 7.4 (Oct. 2010), pp. 337–350. DOI: 10.1109/TDSC.2009.4 (cit. on p. 5).

[72] P. Schwan. "Lustre: Building a file system for 1000-node clusters." In: *Proceedings of the 2003 Ottawa Linux Symposium (OLS 03)*. 2003, pp. 380–386 (cit. on pp. 1, 5, 25).

[73] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. "The Hadoop Distributed File System." In: *Proceedings of the 2010 IEEE Symposium on Massive Storage Systems and Technologies (MSST 10)*. 2010, pp. 1–10. DOI: 10.1109/MSST.2010.5496972 (cit. on p. 15).

[74] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. "Knights Landing: Second-Generation Intel Xeon Phi Product." In: *IEEE Micro* 36.2 (Mar. 2016), pp. 34–46. DOI: 10.1109/MM.2016.25 (cit. on p. 20).

[75] W. Tantisiriroj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross. "On the Duality of Data-intensive File System Design: Reconciling HDFS and PVFS." In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 11)*. 2011, 67:1–67:12. DOI: 10.1145/2063384.2063474 (cit. on p. 1).

[76] Y. Tian, S. Klasky, H. Abbasi, J. Lofstead, R. Grout, N. Podhorszki, Q. Liu, Y. Wang, and W. Yu. "EDO: Improving Read Performance for Scientific Applications through Elastic Data Organization." In: *Proceedings of the 2011 IEEE International Conference on Cluster Computing (CLUSTER 11)*. 2011, pp. 93–102. DOI: 10.1109/CLUSTER.2011.18 (cit. on p. 2).

[77] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. Ø. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, and D. E. Shaw. "A Scalable Parallel Framework for Analyzing Terascale Molecular Dynamics Simulation Trajectories." In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC 08)*. 2008, 56:1–56:12 (cit. on p. 1).

[78] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka. "Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems." In: *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 11)*. 2011, pp. 1–11. DOI: 10.1145/2063384.2063409 (cit. on p. 9).

[79] V. Vishwanath, M. Hereld, and M. E. Papka. "Toward simulation-time data analysis and I/O acceleration on leadership-class systems." In: *Proceedings of the 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV 11)*. 2011, pp. 9–14. DOI: 10.1109/LDAV.2011.6092178 (cit. on p. 9).

[80] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu. "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)*. 2016, 69:1–69:12. DOI: 10.1109/SC.2016.68 (cit. on pp. 9, 21).

[81] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. "Ceph: A Scalable, High-performance Distributed File System." In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 06)*. 2006, pp. 307–320 (cit. on p. 25).

[82] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. "RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters." In: *Proceedings of the 2Nd International Workshop on Petascale Data Storage (PDSW 07)*. 2007, pp. 35–44. DOI: 10.1145/1374596.1374606 (cit. on p. 21).

[83] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. "Dynamic Metadata Management for Petabyte-Scale File Systems." In: *Proceedings of the 2004 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 04)*. 2004, pp. 4–. DOI: 10.1109/SC.2004.22 (cit. on p. 5).

[84] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. "Scalable Performance of the Panasas Parallel File System." In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 08)*. 2008, 2:1–2:17 (cit. on pp. 1, 5, 7, 25).

[85] X. Wu, Y. Xu, Z. Shao, and S. Jiang. "LSM-trie: An LSM-tree-based Ultra-large Key-value Store for Small Data." In: *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 2015, pp. 71–82 (cit. on p. 21).

[86] L. Xiao, K. Ren, Q. Zheng, and G. A. Gibson. "ShardFS vs. IndexFS: Replication vs. Caching Strategies for Distributed Metadata Management in Cloud Storage Systems." In: *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC 15)*. 2015, pp. 236–249. DOI: 10.1145/2806777.2806844 (cit. on p. 6).

[87] J. Xing, J. Xiong, N. Sun, and J. Ma. "Adaptive and Scalable Metadata Management to Support a Trillion Files." In: *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 09)*. 2009, 26:1–26:11. DOI: 10.1145/1654059.1654086 (cit. on p. 5).

[88] J. W. Young. "A First Order Approximation to the Optimum Checkpoint Interval." In: *Commun. ACM* 17.9 (Sept. 1974), pp. 530–531. DOI: 10.1145/361147.361115 (cit. on p. 5).

[89] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. "PreDatA - preparatory data analytics on peta-scale machines." In: *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS 10)*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470454 (cit. on p. 9).

[90] Q. Zheng, G. Amvrosiadis, S. Kadekodi, G. A. Gibson, C. D. Cranor, B. W. Settlemyer, G. Grider, and F. Guo. "Software-defined Storage for Fast Trajectory Queries Using a deltaFS Indexed Massive Directory." In: *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS 17)*. 2017, pp. 7–12. DOI: 10.1145/3149393.3149398 (cit. on pp. 3, 22).

[91] Q. Zheng, C. D. Cranor, D. Guo, G. R. Ganger, G. Amvrosiadis, G. A. Gibson, B. W. Settlemyer, G. Grider, and F. Guo. "Scaling Embedded In-situ Indexing with deltaFS." In: *Proceedings of the 2018 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 18)*. 2018, 3:1–3:15. DOI: 10.1109/SC.2018.00006 (cit. on p. 3).

[92] Q. Zheng, K. Ren, and G. Gibson. "BatchFS: Scaling the File System Control Plane with Client-funded Metadata Servers." In: *Proceedings of the 9th Parallel Data Storage Workshop (PDSW 14)*. 2014, pp. 1–6. DOI: 10.1109/PDSW.2014.7 (cit. on pp. 2, 6, 11, 15).

[93] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider. "DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers." In: *Proceedings of the 10th Parallel Data Storage Workshop (PDSW 15)*. 2015, pp. 1–6. DOI: 10.1145/2834976.2834984 (cit. on pp. 3, 6).