SeqDLM: A Sequencer-Based Distributed Lock Manager for Efficient Shared File Access in a Parallel File System

Qi Chen[1], Shaonan Ma[1], Kang Chen[1], Teng Ma[2], Xin Liu[3], Dexun Chen[3], Yongwei Wu[1], Zuoning Chen[1,4]
Tsinghua University[1], Alibaba Inc[2], National Supercomputing Center in Wuxi[3], Chinese Academy of Engineering[4], China
{chenq17, msn18}@mails.tsinghua.edu.cn, chenkang@tsinghua.edu.cn, sima.mt@alibaba-inc.com
{yyylx, adch}@263.net, wuyw@tsinghua.edu.cn, chenzn@cae.cn

SC22: International Conference for High Performance Computing, Networking, Storage and Analysis | 978-1-6654-5444-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/SC41404.2022.00060

***Abstract*—Distributed locks are used to guarantee the distributed client-cache coherence in parallel file systems. However, they lead to poor performance in the case of parallel writes under high-contention workloads. We analyze the distributed lock manager and find out that lock conflict resolution is the root cause of the poor performance, which involves frequent lock revocations and slow data flushing from client caches to data servers. We design a distributed lock manager named *SeqDLM* by exploiting the sequencer mechanism. *SeqDLM* mitigates the lock conflict resolution overhead using *early grant* and *early revocation* while keeping the same semantics as traditional distributed locks. To evaluate *SeqDLM*, we have implemented a parallel file system called *ccPFS* using both *SeqDLM* and traditional distributed locks. Evaluations on 96 nodes show *SeqDLM* outperforms the traditional distributed locks by up to 10.3× for high-contention parallel writes on a shared file with multiple stripes.**

***Index Terms*—high performance computing, file systems, cache coherence, distributed lock, sequencer**

I. Introduction

Client-cache is widely used in parallel file systems (PFSes) to bridge the gap between high-speed computation and slow storage devices. However, it causes cache coherence problem in the case of concurrent accesses. File systems, such as BeeGFS [1], GlusterFS [2] and Ceph [3], adopt client-cache without concurrency control. This limits their use in high performance computing (HPC) systems, e.g., to support overlapping IO [4], [5] or concurrent producer-consumer workflows [6]. The widely used PFSes in HPC systems, Lustre [7] and GPFS [8], use distributed lock managers (DLMs) [9], [10] to guarantee the client-cache coherence.

In a PFS using DLM, locks granted by lock servers can be cached in clients for future use. This is efficient for accesses that are of spatial locality. For example, for file-per-process (N-N) access pattern [11], data can be cached in clients under the protection of the cached locks. The cached locks are reclaimed by a **lock conflict resolution** when other clients request the same locks. A general conflict resolution process of a write lock is as follows. (1) The lock server sends a revocation message to the client holding the lock (**lock revocation**). (2) The client flushes the dirty data to data servers (**data flushing**), and then (3) releases the lock by sending a release message to the lock server (**lock release**). Only when conflicting locks are totally reclaimed, can a new lock be granted. Thus, for shared file (N-1) access pattern [11], high-contention workloads make DLM almost serialize parallel writes [12].

Both of the N-N and N-1 IO patterns are widely used in HPC. Yang et al. [13] reported that nearly half of applications in Sunway TaihuLight use shared files. To optimize PFSes for high N-1 performance, some works [14]–[16] eliminate DLM and client-cache, but at the expense of degraded N-N performance with small write sizes. In contrast, other studies [12], [17] try to reduce lock conflicts. However, they increase programming complexity and are hard to cope with overlapping IO accesses [5], [18].

IO processing in scientific applications can be typically divided into two phases, a read phase and a write phase, which rarely mixes reads and writes [19]–[21]. Lock conflicts are mainly in the parallel write phase. The heavy conflict resolution of write locks limits the write performance. We propose two methods to mitigate the overhead of lock conflict resolution in the write phase. (1) Usually, the grant of a write lock must wait until the previous lock holder releases the lock. However, concurrent writes do not need to read data from each other. We propose *early grant* to grant the lock before the data flushing of the previous lock holder is completed and use a sequencer mechanism to make the asynchronous data flushing correct (§III-A1). (2) Usually, the revocation of a cached lock is passive, which means that the cached lock is not revoked until another client issues a new conflicting lock request. However, under high contention, locks tend to be transferred to the next holder quickly. We propose *early revocation* (§III-A2), which piggybacks the lock revocation request in the lock grant reply message to release the lock immediately after use, to further eliminate the lock revocation delays for high-contention workloads.

The above proposed methods help DLM to perform better on N-1 writes without sacrificing performance on N-N writes. However, they relax the blocking feature of the traditional write lock, making the modified lock mechanism unable to support atomic write across multiple resources (§III-B1) and atomic read-update operations (§III-B2). **The key challenge here is to keep the same lock semantics while applying the above two optimizations.** Hence, we keep the read lock and extend the traditional write lock to three modes to bridge the semantics gap (§III-C). The rich lock modes provide

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

opportunities to improve performance but bring the complexity of use. We design deterministic lock mode selection rules to ease lock mode selection for common operations in PFSes (§III-C), and adopt an automatic lock conversion mechanism to convert a lock to a proper one when the selected lock mode is no longer optimal (§III-D).

Our major contributions are summarized as follows:

(1). We analyze typical IO patterns in scientific applications and find out the bottleneck of the traditional DLMs. Meanwhile, we validate it with mathematical analysis and micro-benchmarks.

(2). We propose a novel DLM named *SeqDLM* with *early grant* and *early revocation* to eliminate the bottleneck of the traditional DLMs under high contention. Besides, we extend the lock modes to keep the traditional semantics and propose lock mode selection rules and an *automatic lock conversion* mechanism to ease the usage.

(3). We develop *ccPFS* to compare *SeqDLM* with the traditional DLMs on 96 machines. Evaluations show *SeqDLM* achieves up to 18.1× speedup over the traditional DLMs for high-contention parallel writes from 16 clients to a shared file with 1 stripe, and up to 10.3× speedup for high-contention parallel writes from 96 clients to a shared file consisting of 4 stripes, with some writes spanning two stripes.

II. Background and Motivation

A. Distributed Lock in PFS

In a PFS using DLM, files or data objects are associated with *lock resources*. To perform writes, a client first interacts with lock servers to obtain *lock grants* from the lock resources. After the writes are completed, the client can cache the data together with the *lock grants* to reduce network traffic. PFSes usually use a *byte-range locking* mechanism to allow multiple clients to write concurrently to the interleaved parts of a shared file. When a client initiates a *lock request* with a required extent ([start, end]), the server expands the extent range to form a largest compatible address (**lock range expanding**) for reuse in the client. Because applications seldom perform IO backward, only the end of a lock range is usually expanded, as in the implementation of Lustre. We adhere to this convention in this study.

Although DLMs have many variants in practical PFSes [8], [10], the core processing logic of them is similar. We illustrates the process with an example in Fig. 1. In the example, a lock client C0 wants to acquire a write lock with the range [10, 30] while another lock client C1 caching a granted write lock with the range [0, 20]. To begin with, C0 checks the lock grant cache locally. Because no grant is cached, it sends a lock request to the lock server (①). Upon receiving the request, the lock server checks whether the request is *compatible* with locks that have been granted using a *lock compatibility matrix* (LCM) (compatible locks can be granted simultaneously). It finds that a conflicting write lock with the range [0, 20] has been granted and cached in C1. The server then starts a routine of lock conflict resolution. In the conflict resolution, a revocation request is sent to C1. Correspondingly, C1 changes

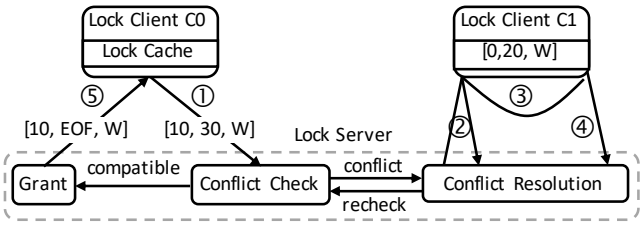

Fig. 1: General processing flow of DLM. W: write lock, ①: lock request. ② lock revocation, ③: data flushing, ④: lock release, ⑤ lock grant.

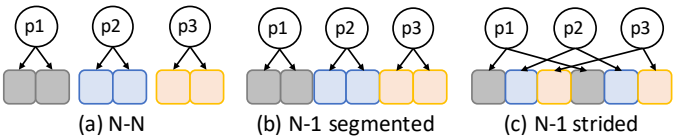

Fig. 2: IO patterns in scientific applications [11].

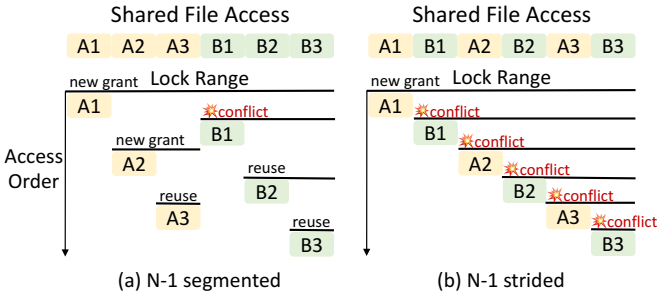

Fig. 3: DLM behaviors in the N-1 patterns. To resolve conflicts, lock server revokes old conflicting locks. To grant a lock, the server expands the end of the lock range to form a largest compatible range.

the lock state to `CANCELING` to prevent future IO from using it and sends a revocation reply to the lock server (②). After that, C1 waits for the ongoing operations to be completed, flushes dirty data to data servers (③), and finally sends a release message to the lock server (④). Upon receiving the release message from C1, the server expands the range of the lock request to [10, EOF] (EOF: End Of File) and grants it to C0 (⑤).

B. Distributed Lock Overhead in Parallel IO

From the perspective of a PFS, scientific applications exhibit two main parallel IO patterns, file-per-process (N-N) and shared file (N-1) [11]. In the N-N pattern (Fig. 2(a)), each file is accessed by one client. In the N-1 pattern, a file is concurrently accessed by multiple clients. The N-1 pattern is further subdivided into N-1 segmented (Fig. 2(b)) and N-1 strided (Fig. 2(c)). In the former, each client accesses a contiguous segment. In the latter, each client accesses non-contiguous segments, which is common in scientific applications [11], [17].

For a PFS using DLM, the client-cache can reduce the write time. In general, write operations return when data are written

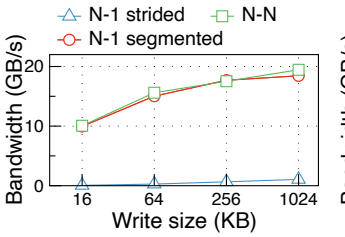

Fig. 4: Performance gap of different IO patterns.

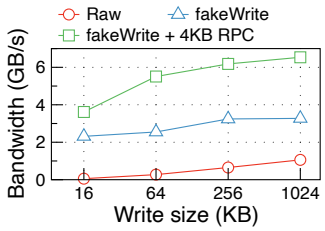

Fig. 5: Performance of reducing data flushing time.

to the cache. The data in the cache are asynchronously flushed to data servers. If data in the cache are needed by other clients (e.g., concurrent scientific workflows), the PFS uses DLM to synchronize the data to data servers before these clients can access them. In the N-N pattern, DLM incurs almost no overhead to IO. In the N-1 segmented and N-1 strided patterns, the overhead of DLM is different. Fig. 3 illustrates DLM behaviors in the N-1 patterns. Here, two clients A and B write to a shared file with the same order. In N-1 segmented, lock conflicts only happen at the beginning of the IO (Fig. 3(a)). While in N-1 strided, the lock range expanding mechanism leads to continual lock conflicts (Fig. 3(b)). We quantify the impact of write lock conflicts on write performance by writing to files located in a 2 GB/s disk array using IOR benchmark [22]. We make this experiment on Lustre (version: 2.10.8), configuring each file to have one stripe and each client to cache at most 2 GB of dirty data. The tests use 16 clients. Each client writes 1 GB data with different write sizes. If no lock conflicts happen, data can be totally cached in clients. As shown in Fig. 4, the write bandwidth of N-N and N-1 segmented is higher and gradually bounded by the cache speed, while the write bandwidth of N-1 strided is much slower. A similar phenomenon exists in other PFSes using DLMs [8]. The gap shows that the traditional DLM is inefficient under high contention.

C. Lock Conflict Resolution Overhead Analysis

We make an analysis to show how the lock conflict resolution limits parallel write performance. We assume that N parallel writes are performed on a shared file with a single stripe. Each write has the size D. These writes are issued by multiple clients and conflict with each other. A typical example of this case is N-1 strided shown in Fig. 3(b). To simplify the analysis, we ignore the overhead of memory operations and assume the network has no software overhead. We take the total data size $N \times D$ divided by the total time spent on the parallel writes as the bandwidth (B_{total}). As shown in Fig. 1, the overhead of acquiring a lock mainly comes from two parts: $p1$ — sending a lock request to the lock server and receiving a grant reply from it, and $p2$ — the lock conflict resolution, including lock revocation, data flushing and lock release. For multiple clients, $p1$ can run in parallel. So the overhead of $p1$ is bounded by OPS (RPC operations per second) of the lock server, and the time of N writes spent on $p1$ can be estimated by $\frac{N}{OPS}$. N conflicting writes lead to $N-1$ **serialized** lock conflict resolutions. Thus, the time spent on $p2$ for N writes can be estimated by $(N-1) \times RTT + \frac{(N-1) \times D}{B_{flush}}$, where RTT is approximately equivalent to the time of lock revocation and lock release (the time spent on revocation reply can be hidden by the lock release RPC), and B_{flush} is the bandwidth of data flushing. Hence, B_{total} can be estimated by Equation (1). That means, the total bandwidth of N conflicting writes is limited by ①$\frac{1}{OPS \times D}$, ②$\frac{RTT}{D}$ and ③$\frac{1}{B_{flush}}$. B_{flush} is decided by B_{net} (bandwidth of network) and B_{disk} (bandwidth of disk). It can be estimated by Equation (2).

TABLE I: Approximate performance of commonly-used Infiniband and NVMe SSD.

OPS (op/sec.)	RTT (sec.)	B_{net} (Byte/sec.)	B_{disk} (Byte/sec.)
1×10^7	1×10^{-6}	12.5×10^9	3×10^9

$$B_{total} = \frac{N \times D}{\frac{N}{OPS} + (N-1) \times RTT + \frac{(N-1) \times D}{B_{flush}}} \approx \frac{1}{\frac{1}{OPS \times D} + \frac{RTT}{D} + \frac{1}{B_{flush}}} \quad (1)$$

$$B_{flush} \approx \frac{D}{\frac{D}{B_{net}} + \frac{D}{B_{disk}}} = \frac{B_{net} \times B_{disk}}{B_{net} + B_{disk}} \quad (2)$$

For further explanation, we assume $D = 10^6$ bytes (≈ 1 MB) and use parameters in Table I to evaluate ①, ② and ③. The result is that ① $\approx 1.0 \times 10^{-13}$ sec/bytes, ② $\approx 1.0 \times 10^{-12}$ sec/bytes and ③ $\approx 4.1 \times 10^{-10}$ sec/bytes. ③ is much larger than ① and ②. From the above analysis, we conclude that ③ is the bottleneck of DLM under high contention, and verify it by gradually reducing data flushing overhead in Lustre with the same configuration in §II-B. We achieve this by (1) disabling disk write using fakeWrite [23] and (2) hacking Lustre to only transfer the first page (4KB) if a flushing RPC contains multiple pages. Fig. 5 shows that reducing data flushing overhead can greatly improve performance. In this test, we also find that many lock requests are queued on the lock server due to contention, and many granted locks are revoked immediately. From Equation (1), we can see that if we can eliminate the overhead of data flushing (removing ③), the lock revocation (contributing to ②) becomes a new bottleneck.

In order to increase IO parallelism and exploit the performance of multiple devices, PFSes usually organize files into multiple stripes and each stripe is associated with a dedicated lock resource. For this case, if no single write spans multiple stripes, the lock conflict resolution only affects conflicting writes in a single stripe. Otherwise, writes across multiple stripes need to obtain multiple locks from the lock resources of the corresponding stripes, and the lock conflict resolution will decrease the write parallelism of multiple stripes.

III. *SeqDLM* Design

The basic idea of *SeqDLM* is to leverage the sequencer mechanism to accelerate the write-write lock conflict resolution in DLM. In this section, we first present the basic idea

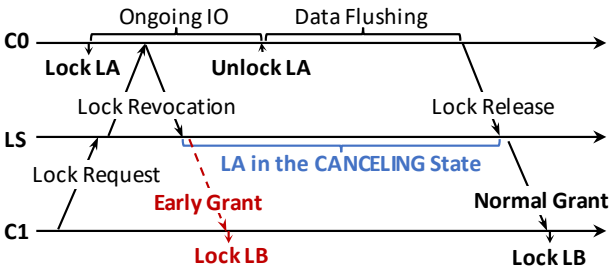

Fig. 6: Normal grant vs. early grant. C0, C1: clients. LS: lock server. Write lock LB from C1 conflicts with a granted write lock LA that is cached in C0.

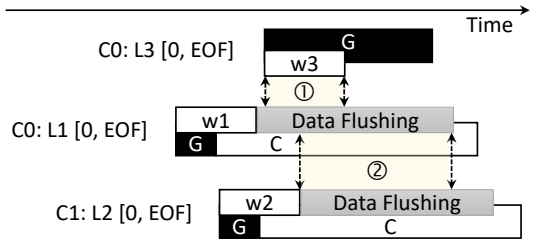

Fig. 7: Data safety problems of early grant without a sequencer. C0, C1: clients. w1, w2, w3: write operations. L1, L2, L3: write locks. G: lock in the GRANTED stat. C: lock in the CANCELING state. Lock grant order: L1 → L2 → L3. The lock server expands the end of each granted lock to EOF according to the lock range expanding strategy.

of *early grant* (§III-A1) and *early revocation* (§III-A2). We then discuss the semantics gap (§III-B) between the basic idea and the traditional DLM. Next, We outline lock modes of *SeqDLM* to bridge the gap and lock selection rules to ease the use (§III-C). Finally, we propose an automatic lock conversion mechanism to convert a lock to a proper mode for high performance when the selected lock mode is no longer optimal (§III-D).

A. Basic Idea

1) Early Grant: In the traditional DLM, when conflicting with a granted lock, a new grant should wait until the granted lock is released (*normal grant* shown in Fig. 6). It is needed to resolve the conflict between a read lock and a write lock because data dependency may exist between them. However, for parallel writes, which do not read data from each other, waiting for conflicting writes to be fully flushed to data servers is too heavy. *Early grant* is proposed to enable a write lock to be granted earlier (*early grant* shown in Fig. 6): upon receiving the revocation reply from C0, knowing that the old write lock LA granted to C0 has been changed into the CANCELING state and will not be reused, the lock server directly grants the new write lock LB to C1.

Fig. 7 illustrates that three write operations (w1, w3 from C0 and w2 from C1) are performed in turn on a shared file with *early grant*. Each write has the range [0, 1GB]. *Early grant* causes two data safety problems. ① w3 may start but the data flushing of w1 has not been completed, resulting in data corruption in the client-cache. ② Data flushing of w1 and w2 may reach the data servers out of order, breaking write serialization. We address these problems by leveraging the sequencer mechanism. Each lock resource maintains a monotonically increasing sequence number (SN). When granting a lock, the lock server assigns the current SN of the lock resource to the lock. If the lock to be granted is a write lock, the lock server also increases the lock resource's SN by one. In this way, all granted write locks of a lock resource can be ordered by their unique SNs. Data to be written are tagged with the SNs from the corresponding granted locks. When adding them to the client-cache, we use the SNs to distinguish which data are the newest, thus addressing the problem ①. Data that are flushed to data servers also carry their SNs. The data server keeps an extent cache to track the SNs of data it has processed. Leveraging the SN, the server can correctly write the data to storage devices, thus addressing the problem ② (referring to §IV-B for details).

Ordering in *SeqDLM* is different from other systems depending on the same mechanism. For example, CORFU [24], a distributed shared log abstraction, uses a global sequencer to order concurrent writes. However, each write in CORFU needs to get a SN, which makes the sequencer become a bottleneck. In *SeqDLM*, if a lock grant is cached in a client, the SN of it can be reused by subsequent writes. Thus, under low contention, ordering in *SeqDLM* incurs low overhead to lock servers.

2) Early Revocation: According to the analysis in §II-C, after *early grant* removes the data flushing from the critical path of the lock conflict resolution, the lock revocation overhead cannot be ignored under high contention. For example, conflicting lock requests (LA, LB) from two clients reach the lock server simultaneously. After sending the grant reply to LA, the server finds LB conflicts with LA and immediately sends a revocation callback to LA. The grant of LB has to wait for at least one RTT until the revocation reply of LA has returned.

We propose *early revocation* to eliminate this delay. When granting a lock, the lock server tags it with a state to tell the client how to handle the lock after use. A GRANTED state means the lock can be cached and reused by the client, while a CANCELING state means the lock cannot be reused and should be canceled after use. Under high contention, lock requests are queued and the lock server can detect whether a lock to be granted conflicts with future locks in the queue. If it does, we piggyback the revocation request in the grant reply by tagging the lock with the CANCELING state. After that, the lock server can safely grant new locks without waiting for the revocation replies to return. After eliminating the wait, combinded with *early grant*, the lock server acts like a global sequencer in high-contention write scenarios.

The lock server uses *early revocation* for a lock to be granted when (1) it detects that a newer lock request conflicts

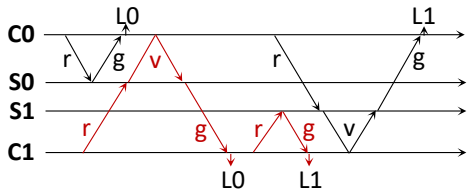

Fig. 8: Anomaly of writes across two stripes due to early grant. r: lock request, g: lock grant, v: lock revocation.

with the lock, and (2) it cannot expand the lock range. The reason for (2) is as follows. The lock server always expands the real range a client requires. Locks that cannot be expanded are hardly reused because clients rarely write to the same file range repeatedly. Caching these locks in clients incurs the overhead of maintaining lock states [25].

B. Semantics Gap with Traditional DLM

1) Atomic Write across Multiple Resources: PFS usually organizes a file into multiple stripes and each stripe has a lock resource. When a write spans multiple stripes, the client should first hold write locks of the stripes. In this case, *early grant* cannot guarantee the write atomicity. We take an example to explain the anomaly. A file has two stripes located in two servers S0 and S1. The stripe size is 1MB. Two clients C0 and C1 want to write to the file with the range [0, 2MB], which spans S0 and S1. To ensure write atomicity, each client needs to acquire write locks (L0 and L1, respectively) from S0 and S1. The L0 from C0 is first granted by S0. Using the traditional write lock, L0 from C1 is blocked until C0 finishes the write and releases L0. However, *early grant* relaxes the blocking feature. As shown in Fig. 8, the grant of L0 to C1 cannot be blocked, which may cause L1 from C1 to be granted before L1 from C0. As a result, the final content of [0, 1MB] is from C1 and that of [1MB, 2MB] from C0, which breaks the atomicity of a single write operation.

2) Atomic Read-update Operations: Some write operations in a PFS are accompanied by read operations, called atomic read-update operations. For example, the `append` operation needs to read the global file size before it can perform the write. Besides, PFSes usually use a 4KB page as the minimal management unit. In the implementation of most PFSes, a partial page write needs a synchronous page read and then an update on the page. When adopting *early grant*, a client may get a write lock grant but dirty data belonging to old conflicting write locks have not been flushed to data servers. This makes the client fail to obtain accurate file size or up-to-date data.

C. Lock Modes and Lock Mode Selection

In order to bridge the gap mentioned in §III-B, we keep the traditional read lock and refine the traditional write lock into three modes. These lock modes are described as follows.

PR (Protective Read). Holders of PR can read the shared resource simultaneously, which has the same semantics as the traditional read lock.

TABLE II: LCM of *SeqDLM*. ①: lock mode of a new lock request, ②: lock mode of a granted lock. Y: compatible. N: incompatible. **N/Y** means that the lock request is incompatible with the granted lock in the `GRANTED` state while it is compatible with the lock in the `CANCELING` state.

① \ ②	PR	NBW	BW	PW
PR	Y	N	N	N
NBW	N	**N/Y**	N	N
BW	N	**N/Y**	N	N
PW	N	N	N	N

NBW (Non-blocking Write). The holder of NBW can only write the shared resource but is not allowed to read it. Besides, it does not maintain the blocking feature of the write lock. NBW delivers high parallel write performance under high contention.

BW (Blocking Write). BW is similar to NBW but keeps the **blocking** feature of the write lock. BW is used to bridge the gap in §III-B1.

PW (Protective Write). PW has the same semantics as the traditional write lock. The holder of PW can read and write the shared resource. PW is used to bridge the gap in §III-B2.

SeqDLM has a processing logic similar to the traditional DLMs. The main difference is that we define a new LCM to check conflicts between the new lock modes. In *SeqDLM*'s LCM (shown in Table II), a granted lock has two states, `GRANTED` and `CANCELING`, which are mentioned in §III-A2. A granted lock is in the `GRANTED` state by default. It enters into the `CANCELING` state when the revocation reply of it has been processed by the lock server or the lock has been granted with *early revocation*. If a new lock request with the NBW or BW mode conflicts with granted NBW locks in the `GRANTED` state, once the granted locks enter into the `CANCELING` state, the conflicts are resolved, and then the new lock request can be granted (*early grant*). For other cases, lock conflicts are resolved only when the conflicting granted locks are completely released (*normal grant*).

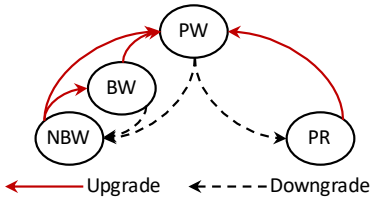

Fig. 9: Lock mode severity and conversion in *SeqDLM*.

More write lock modes bring more opportunities for performance improvement. However, selecting a proper lock mode is essential. To simplify the selection, similar to prior works [9], [26], we order the lock modes by the severity of their restrictiveness, as shown in Fig. 9. Lock modes in the upper layers are more restrictive than that in the lower layers. A more restrictive lock can be used in more scenarios. For example, PW, the most restrictive lock mode in *SeqDLM*, can

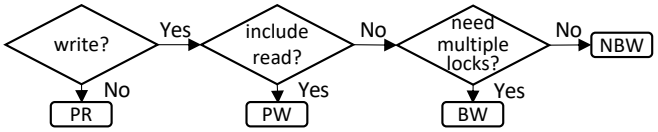

Fig. 10: Lock mode selection rules for an IO operation.

be used to support writes across multiple resources and atomic read-update operations, while the least restrictive write lock NBW can only be used for writes in a single resource. On the other hand, a more restrictive lock is less efficient under lock contention. For example, PW and BW serialize the data flushing of conflicting locks, while NBW can leverage *early grant* and *early revocation* to get high IO performance under high contention.

We give the rules, shown in Fig. 10, to select an optimal lock mode for each operation. Here, the optimal lock mode means the least restrictive lock mode that meets the need of an IO operation. For a read operation, PR is selected. For a write operation, if it may involve implicit reads (e.g., *append*), PW is selected. Otherwise, if it needs to hold multiple locks simultaneously (e.g., atomic write across multiple resources), BW is selected. For other cases, NBW is selected. For common data-related operations in the file system, such as *read*, *write*, *truncate* and *append*, deterministic lock modes can be easily selected according to the rules.

D. Automatic Lock Conversion

For an operation sequence, selecting an optimal lock mode for each operation does not mean it is optimal for the whole sequence. We propose an *automatic lock conversion* mechanism (lock upgrading and downgrading) to convert a lock to a proper mode for high performance when the selected lock mode is no longer optimal. This mechanism is automatically triggered by the lock service and transparent to users.

1) Lock Upgrading: Fig. 11(a) shows the lock behavior of mixed reads/writes from the same client to a shared file with one stripe. According to the lock mode selection rules in Fig. 10, the write operation selects NBW and the read operation selects PR. PR and NBW conflicts in *SeqDLM* (Table II). This leads to frequent lock revocations in the same client. We use *lock upgrading* to address the problem. Once the lock server finds that a lock request conflicts with a granted lock in the same client, it grants the lock with a more restrictive lock mode. As shown in Fig. 11(b), the lock server upgrades the mode of the lock request from PR to PW and grants it to the client. When receiving the lock grant, the client merges the old conflicting NBW lock in its local cache with the new granted PW lock. In this way, we avoid reclaiming locks due to lock conflicts in the same client. In addition, subsequent reads/writes can also reuse the granted PW lock. The possible *lock upgrading* routines are shown in Fig. 9. If a lock mode needs to be upgraded to PW while multiple clients are caching conflicting PR locks, the lock server first reclaims the PR locks from the clients except the one that sends the lock request.

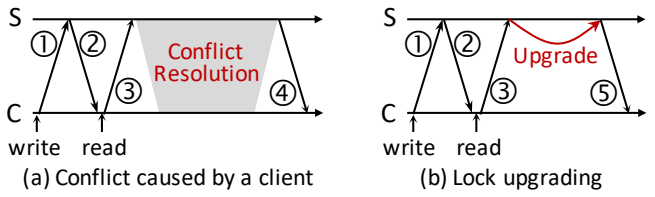

Fig. 11: Comparison before and after using *lock upgrading*. C: client, S: server, ①: NBW request, ②: NBW grant, ③: PR request, ④: PR grant, ⑤: PW grant.

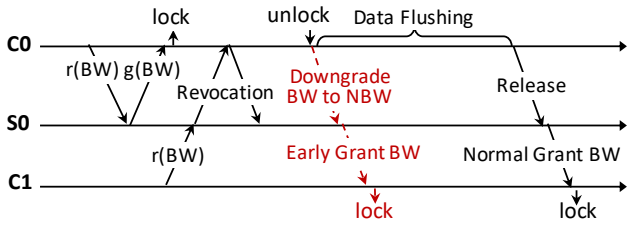

Fig. 12: Lock downgrading. r: request. g: grant.

2) Lock Downgrading: As shown in Table II, a lock request, regardless of its mode, is incompatible with a granted BW or PW lock in both `GRANTED` and `CANCELING` states. It means that once a client has selected BW or PW, or a lock has been upgraded to BW or PW, a new conflicting lock grant needs to be delayed until the BW or PW locks are released. For example, two clients C0 and C1 write to a file with two stripes located in two servers S0 and S1. Each write spans the two stripes. Fig. 12 shows the lock processing on S0. Each client selects BW according to our lock selection rules. Due to the blocking feature of BW, the lock grant to C1 should wait until data flushing in C0 is completed (*normal grant*). However, this delay is unnecessary because C0 and C1 do not perform read. We use *lock downgrading* to address this problem. When a client starts to cancel a lock, it first downgrades the lock to the least restrictive one. As shown in Fig. 12, after the BW lock is unlocked, finding the lock is in the `CANCELING` state, C0 downgrades it to a NBW one and sends a downgrading RPC to S0 to notify the lock downgrading. After that, C0 continues the lock canceling process. Upon receiving the downgrading RPC, S0 changes the lock to a NBW one. Then the BW lock request from C1 can be granted using *early grant*.

The possible downgrading routines are shown in Fig. 9. If a PW lock in the `CANCELING` state is held by only readers, the client flushes the dirty data and then downgrades it to a PR lock. After that, PR lock requests conflicting with the PW lock can be granted. For other cases, the client downgrades the PW lock to a NBW one so that new NBW or BW lock requests conflicting with it can be early granted.

IV. *ccPFS*: A *SeqDLM*-BASED PFS

We develop *ccPFS* to show the essentials of applying *SeqDLM* in a PFS. *ccPFS* is used as a client-cache coherent burst buffer system. The architecture is shown in Fig. 13. It

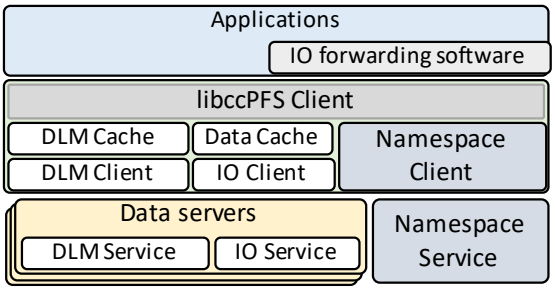

Fig. 13: *ccPFS* architecture.

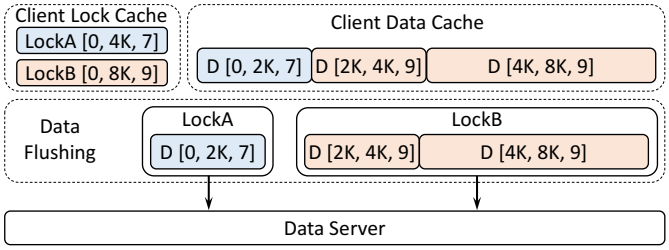

Fig. 14: Client-side write routine in *ccPFS*.

uses an external distributed file system, such as Lustre or NFS, to provide metadata service. It organizes a file into one or more stripes. Each stripe is associated with a unique lock resource. Each stripe and its corresponding lock resource have the same identifier (ID). *ccPFS* distributes stripes to data servers by hashing the IDs. Each data server runs an IO service to handle IO requests to file stripes, and a DLM service to handle lock requests to the corresponding stripes. *ccPFS* allows data to be cached in clients and leverages *SeqDLM* to guarantee the client-cache coherence.

ccPFS uses CaRT [27] as the RPC infrastructure for *SeqDLM*, and uses InfiniBand Verbs to transfer IO requests between clients and servers. To avoid memory registration overhead, each client allocates a certain number of pages as a memory pool and registers all of these pages as RDMA-enabled space in advance. All cached pages in a client are picked up from this memory pool. When the number of cached pages exceeds a threshold, the client reclaims some cached pages to the pool. *ccPFS* provides a library called *libccPFS* to provide POSIX-like APIs. It can be linked into applications for direct use, or be implemented as the back-end of an IO forwarding software. For the latter, applications do not need to modify their codes to use *ccPFS*.

Similar to Lustre, the locking operations of *SeqDLM* are implicitly included in IO operations and transparent to application users. For example, when a write operation is called, *ccPFS* automatically obtains a lock (*lock()*) with the file range it will write to. When the write is done, it puts the lock (*unlock()*). After that, the lock can be cached in the clients. *ccPFS* uses the lock range expanding mechanism so that a granted lock can be reused by a client as much as possible. The lock modes are selected according to the lock selection rules in Fig. 10. *ccPFS* asynchronously flushes conflicting writes and uses the SN in *SeqDLM* to guarantee correctness. In this section, we describe how to use SN in *SeqDLM* to resolve client-cache conflict (§IV-A) and make data flushing correct on data servers (§IV-B). We end this section with a discussion of durability and handling failures (§IV-C).

A. Resolving Client-Cache Conflict

In the client, cached data is divided into pages (e.g. 4KB). Each page maintains an extent list to keep SNs of valid data blocks in it. Fig. 14 shows the path of two conflicting writes in a client. A write with the extent [0, 4KB] has been performed under lockA (lockA is in the `CANCELING` state) while another write with the extent [2KB, 8KB] is performed under lockB. Written data with a larger SN overwrites the smaller ones when inserted into the client-cache. When revoking a lock, the client flushes data with the same SN of the lock. For example, for LockB, the client flushes D [2K, 4K, 9] (data with the range [2KB, 4KB] and SN 9) and D [4K, 8K, 9]. Data flushing of multiple locks can be batched in one RPC and the RPC carries the SNs of involved data blocks.

B. Making Data Flushing Correct

Data flushing of conflicting locks may arrive at the data server out of order. We use an *extent cache* to make the conflicting data flushing correct. **The extent cache records the maximum SNs of data blocks that have been written to devices.** The entries in the extent cache for each stripe are organized using an interval tree. Each entry consists of an extent and its newest SN and has a size of 48 bytes. Continuous extents of the same stripe with the same SN are merged to reduce extent cache size. Fig. 15 illustrates the write routine on the data server. A write request, carrying three data blocks (D [0, 2K, 7], D [2K, 4K, 9] and D [4K, 8K, 9]), arrives. First, we merge the SNs into the extent cache (①). For the overlapping part, we keep the extent with a larger SN. For example, the SN of the extent [0, 2K] is not changed because D [0, 2K, 7] is older than D [0, 4K, 8], while the SN of the extent [2K, 4K] is updated because D [2K, 4K, 9] is newer than D [0, 4K, 8]. During the merging, we record the changes in an *update set* (②). Next, we write the data in the update set to devices and discard the parts that are not in the set (③). Optionally, we can maintain an *extent log* for each stripe and record the entries in the update set into the log to enable the extent cache to be reconstructed from the log in the case of server recovery (§IV-C2) (④). Finally, we send a reply to tell the client that data flushing has been completed.

For contiguous writes, such as in the N-N and N-1 segmented patterns, the extent entries can be merged, resulting in a smaller extent cache size. However, when serving lots of non-contiguous writes, such as in the N-1 strided pattern, the extent cache may have a large size, which potentially degrades write performance. We use the following methods to keep the extent cache at a moderate size. (1) When the total number of cached entries exceeds a threshold (256 K), the server starts an asynchronous task to clean up invalid entries. For each stripe in the cache, the task picks up some entries, queries

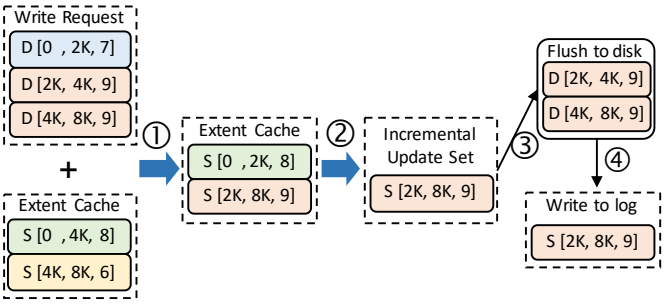

Fig. 15: Server-side write routine in *ccPFS*. S [a, b, SN]: the sequence of the extent [a, b] is SN.

the minimum SN (mSN) of unreleased write locks of the corresponding stripes whose ranges overlap the entries, and removes the entries whose SNs are no larger than the mSN. For example, as shown in Fig. 15, if we find the mSN of the extent [0, 8K] is 9, we know that no more data blocks with smaller SNs than 9 need to be processed because *SeqDLM* ensures data with SNs smaller than mSN have been written to devices. Then we can safely remove S [0, 2K, 8] and S [2K, 8K, 9]. The task runs periodically with a lower priority than IO threads and only processes at most 1,024 entries each time to prevent it from blocking normal IO. Most entries can be cleaned in this case. (2) If the number of cached entries that cannot be cleaned in (1) still exceeds the threshold, it means new write locks have been early granted but many old granted locks in the `CANCELING` state due to *early grant* have not completed their data flushing. The server then forcefully synchronizes data flushing of all clients by requiring a read lock with the whole range of each stripe. If an extent log of a stripe is maintained, it can be safely removed in this case.

C. Discussion

1) Durability: Client-cache causes data loss when a client crashes while there are dirty data in it. This convention is applicable to local file systems, such as ext4 and XFS, and distributed file systems, such as BeeGFS [1] and Lustre [7]. Nonetheless, we try to reduce data loss using the following best-effort strategy. We set the minimum and maximum threshold for the total size of the client-cache. When the size of dirty data reaches the minimum threshold (256 MB by default), the client starts a daemon to voluntarily flush the dirty data. When the size of dirty data reaches the maximum threshold (4 GB by default), the client forcefully flushes the dirty data and blocks new writes until there are available caches. Besides, *ccPFS* also provides *fsync()* to allow users explicitly to flush dirty data if needed.

2) Server Recovery: When failures happen, HPC applications are usually aborted and restarted. Thus, *ccPFS* does not provide the server recovery mechanism, similar to other ephemeral PFSes, such as BurstFS [15] and GekkoFS [14]. However, *SeqDLM* has the ability to handle server recovery similar to the traditional DLMs when applied to other systems. First, the server recoveries lock states by gathering them from all clients. Second, it replays the *extent log* to reconstruct the extent cache (the extent log should be maintained for server recovery). Finally, the clients redo the data flushing RPCs that were sent but the replies have not been received because of failures.

V. Evaluation

In this section, we aim to answer the following questions.
(1). How does *SeqDLM* perform under high contention?
(2). How does *SeqDLM* perform to support HPC workloads using synthetic benchmarks and application benchmarks?

A. Experimental Setup

In order to easily track performance metrics, we use *ccPFS* as the basic platform and implement traditional DLMs in *ccPFS*. They include the general DLM (DLM-basic) shown in §II-A, Lustre-special DLM (DLM-Lustre), and datatype locking (DLM-datatype) [17]. *SeqDLM*, DLM-basic and DLM-Lustre use extents to manage lock ranges. The lock server of them can expand the range of a granted lock. The difference is that *SeqDLM* and DLM-basic greedily expand the end of a granted lock to a largest compatible range while DLM-Lustre expands the lock range to a maximum of 32 MB when the server finds that more than 32 locks have been granted. The special optimization of DLM-Lustre helps to reduce lock conflicts under high contention. DLM-datatype is an optimized implementation to support atomic non-contiguous IO. It uses a datatype to describe lock ranges of non-contiguous IO to reduce lock request size and the lock server of it does not expand lock ranges to reduce lock conflicts.

We run our experiments on a 96-node cluster. Each node has two 8-core intel Xeon Silver 4214 CPU 2.2 GHz, equipped with 256 GB RAM, one 3200 GB NVMe SSD and a 100 Gbps HDR NIC connected to a HDR IB switch. The nodes run CentOS 7.7 with Linux 3.10.0-862.14.4 kernel. We use CaRT [27] from DAOS 1.1.2 [16] for RPCs in *SeqDLM*, which is based on Mercury 2.0.1 [28]. CaRT runs on $verbs; ofi\_rxm$ NA plugin, achieving about 213 kOPS for a server. We use IB verbs for IO between clients and data servers in *ccPFS* and use MPICH to support distributed tests.

B. Micro Benchmark

1) Data Safety: For read-write conflicts, *SeqDLM* and the traditional DLM have the same behaviors. However, for write-write conflicts, *early grant* and the automatic lock conversion mechanism may lead to conflicting data flushing. We use some workloads to test data safety in the write-write conflict scenarios. First, we use the IO500 [29] IOR hard workload to verify the data safety of parallel non-overlapping writes. It first writes to a shared file using N-1 strided pattern with the write size 47,008 bytes, and then reads them back from different clients to check whether the result is correct. We test *ccPFS* with the stripe number 1, 2 and 4 on 16 clients. We run the test for 10 times and observe that *ccPFS* always returns correct results. Second, we build a workload, as shown in Fig. 7, to show that *ccPFS* is data safe under parallel overlapping

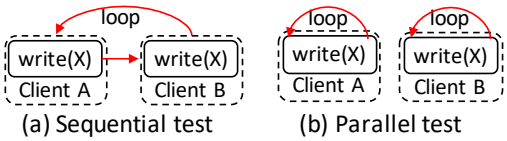

Fig. 16: Experimental setup for *SeqDLM*. X: write size. To ensure that locks among clients are conflicting, each write acquires a write lock with the range [0, EOF].

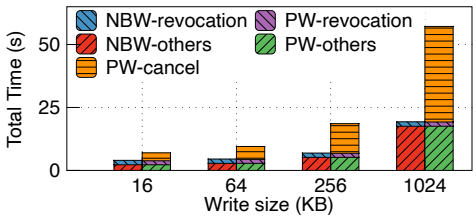

Fig. 17: Time breakdown in the sequential test.

writes. We first run concurrent writes on 16 clients to a shared file. Each client performs two writes with different data. Each write has the range [0, 1GB]. When all writes are completed (guaranteed by *MPI_Barrier()*), each client performs a read with the range [0, 1GB]. We compare the checksum of the data to see whether they are the same. We repeat the test for 10 times for a file with 1 stripe (to test NBW) and with 2 stripes (to test BW with lock conversion). We observe the results are the same and from the second write of some client in the test. This is consistent with the traditional lock semantics.

2) Early Grant and Early Revocation: We compare NBW with PW in *SeqDLM* to show the impact of *early grant* and *early revocation* on IO performance.

Overhead breakdown. We explain how *early grant* improves IO performance by measuring the time of a totally conflicting write sequence, in which 16 clients write to a shared file alternately in a round-robin fashion (Fig. 16(a)). `MPI_Send()/MPI_Recv()` are used to synchronize operations between clients. Each client performs 4,000 writes. We break down the total time into three parts: ① lock revocation, ② lock cancel (including data flushing and lock release RPCs) and ③ others (including lock requests, grant replies and writing data into the client-caches). We measure the time spent on each part. The results are shown in Fig. 17. For writes using PW, the lock conflict resolution (① + ②) is the most time-consuming part (67.9% to 69.3% of the total time for X from 16KB to 1,024KB). The time is mainly spent on ② (66.5% to 95.7% of the locking time for X from 16KB to 1,024KB), which is bounded by the data flushing. The time spent on the data flushing increases as X increases. Because *early grant* decouples the data flushing from the lock conflict resolution, the total time using NBW is much less than that using PW.

Throughput. We use parallel writes from 16 clients to evaluate the throughput of one lock resource under high contention. Each client independently writes to a shared file for 4,000 times (Fig. 16(b)). We divide the total number of writes (64,000) by the time spent in the test as the throughput.

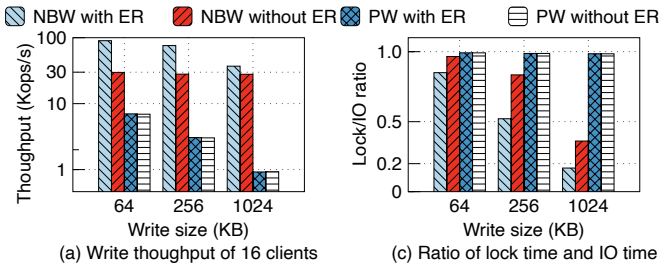

Fig. 18: *SeqDLM* parallel test results. ER: early revocation.

In this test, multiple lock requests may arrive at the server simultaneously. Therefore, the server has opportunities to use *early revocation*. We also test the effect of *early revocation* on NBW and PW. Fig. 18(a) shows the results. The throughput of PW is not affected by *early revocation*. The throughput of NBW (*early grant*) without *early revocation* outperforms PW (*normal grant*) by 4.26× and 30.29× when the write size is 64KB and 1,024KB, respectively. The throughput of NBW with *early revocation* outperforms PW by 12.88× and 40.18× when the write size is 64KB and 1,024KB, respectively. In order to better understand the performance, we record the ratio of the locking time and IO time (locking/IO ratio) on one client (shown in Fig. 18(b)). *Early grant* decouples data flushing from lock conflict resolution, which reduces the locking time. Thus, as the write size increases, the locking/IO ratio decreases for NBW. *Early revocation* avoiding unnecessary wait for lock revocation replies further reduces the locking time.

3) Lock Conversion: **Lock upgrading.** We evaluate the benefit of lock upgrading using 1,000 interleaved writes and reads from a client on a file with 1 stripe. In the test, the read operations use PR while the write operations can use PW or NBW. Fig. 19(a) shows the throughput of the test. If the write operations use PW, after the first write gets a PW lock, the subsequent reads and writes can reuse the lock, thus having a high throughput. If the write operations use NBW without the lock conversion, the operation sequence causes continuous lock conflicts. With lock conversion, NBW is upgraded to PW when the server handles the first conflict. After that, reads and writes can reuse the PW lock. Thus, it has the similar performance to PW.

Lock downgrading. We evaluate the benefit of lock downgrading using parallel writes to a file with two stripes from 16 clients. Each write spans the two stripes and should hold write locks from both stripes, simultaneously. PW and BW can support this scenario. Fig. 19(b) shows the results. Without the lock conversion, BW and PW have similar performance. With lock conversion, BW outperforms PW by 2.48× and 9.40× with the write size of 64K and 1,024K, respectively. That is because lock conversion automatically downgrades BW to NBW during the lock conflict resolution. After that, a BW lock request conflicting with the old BW can be granted using *early grant*.

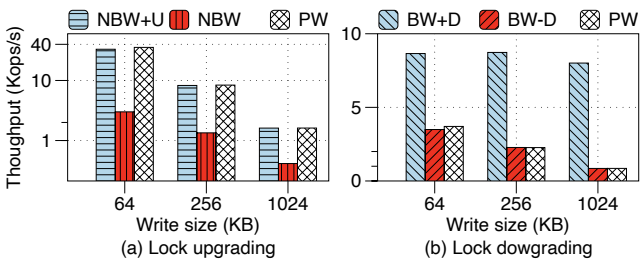

Fig. 19: The benefits of lock conversion. NBW+U: NBW with lock upgrading, BW+D: BW with lock downgrading, BW-D: BW without lock downgrading.

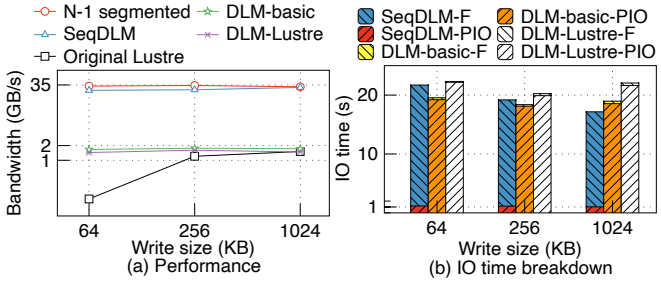

Fig. 20: IOR results on a file with 1 stripe.

C. IOR Benchmark

IOR benchmark reflects HPC IO requirements in broad HPC workloads [30]. We use it to generate typical HPC IO workloads to compare *SeqDLM* with traditional DLMs in *ccPFS*. Each data server of *ccPFS* has a NVMe SSD formatted with XFS as the back-end storage. Each client is configured to cache at most 4GB dirty data and has a daemon to asynchronously flush dirty data to servers using the best-effort strategy presented in §IV-C1. During the test, each client writes 2GB data. We record the time spent on the write (IOR without fsync flag) as parallel IO (PIO) time and **calculate the bandwidth using the PIO time**, which represents the write performance that applications can see. At the end of each test, we flush the dirty data to data servers and record the time spent on it as the flushing (F) time.

1) Performance of a File with a Single Stripe: We use the workload of IOR N-1 segmented with the write size 64 KB to evaluate the performance of different DLMs on a single-striped file under low contention. The test runs on 16 clients. Table III shows the results. The bandwidth of the three DLMs is similar. Besides, the total IO time (including the PIO time and F time) is also similar. This indicates that *SeqDLM* maintains the advantage of the traditional DLMs, and the ordering in *SeqDLM* does not impose too much overhead on data flushing under low contention.

TABLE III: Results of IOR N-1 segmented on a file with 1 stripe.

DLMs	SeqDLM	DLM-basic	DLM-Lustre
Bandwidth (GB/s)	33.2	33.8	33.7
total IO time (sec.)	18.1	19.1	19.5

We use the workload of IOR N-1 strided to evaluate the performance of different DLMs under high contention. The test also runs on 16 clients. For comparison, we include the performance of N-1 segmented using SeqDLM. Besides, we also evaluate N-1 strided performance on Lustre (original Lustre) with the same hardware configuration. The Lustre version is 2.10.8 and each Lustre client is configured to cache at most 2GB dirty data.

The results are shown in Fig. 20(a). DLM-Lustre in *ccPFS* has better performance with smaller write size than that in original Lustre because of the registered memory pool presented in IV. The gap is becoming smaller as the write size increases. With the write size 1,024KB, the bandwidth of the traditional DLMs is bounded by the maximum performance of the device. N-1 strided using *SeqDLM* achieves 81.7% to 96.9% performance of N-1 segmented when the write size is from 64KB to 1,024KB, and gains up to 18.1× speedup than DLM-basic and DLM-Lustre. That is because *SeqDLM* selects NBW for writes on a single-striped file. *Early grant* and *early revocation* greatly reduce the locking time.

Fig. 20(b) shows the total IO time of each test. The total time is bounded by the performance of the storage device in the data server. However, the PIO time using *SeqDLM* takes about 5% of the total time because it decouples data flushing from lock conflict resolution. The PIO time using DLM-basic and DLM-Lustre takes up to 99% of the total time because a new lock can be granted only when old conflicting locks have been released and the data flushing of old locks accounts for a large proportion of the locking time. This makes a significant performance difference between *SeqDLM* and the traditional DLMs. It also implies that *SeqDLM* can make the best of client-cache even under high contention.

2) Performance of a File with Multiple Stripes: We use N-1 strided with the write sizes 47,008 bytes (IO500 IO-hard write benchmark [29]), 188,032 (47,008 × 4) bytes and 752,128 (47,008 × 16) bytes on a shared file with 4 and 8 stripes to show the *SeqDLM* efficiency. The stripe size is 1 MB. These writes are not 4KB aligned. However, *SeqDLM*, DLM-basic and DLM-Lustre align lock ranges with 4KB. Thus, adjacent writes conflict with each other. Besides, some writes in the test span two stripes.

As shown in Fig. 21, the IO bandwidth using DLM-basic and DLM-Lustre increases as the write size increases because a larger write size is favorable to devices. DLM-Lustre has better performance than DLM-basic because it limits the range expanding under high contention, resulting in fewer lock conflicts. The performance using DLM-basic and DLM-Lustre is gradually bounded by the performance of storage devices because of data flushing overhead. In contrast, the IO performance using *SeqDLM* increases as the write size increases, which is not limited by the performance of storage devices. The reason is that for writes inside one stripe, *ccPFS* using *SeqDLM* selects NBW, which decouples lock revocation and data flushing from the lock conflict resolution, while for

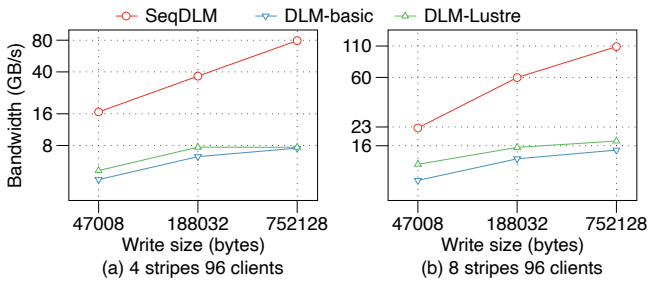

Fig. 21: N-1 strided performance on a file with multiple stripes.

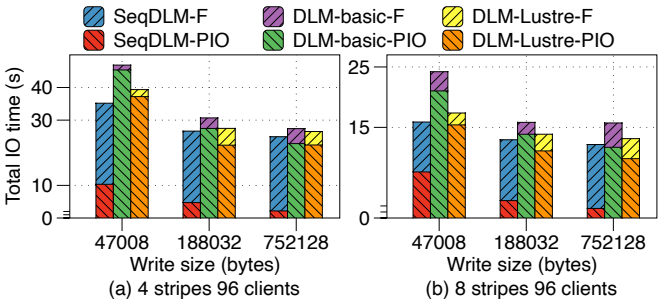

Fig. 22: Total IO time of N-1 strided on a file with multiple stripes.

writes across multiple stripes, although *ccPFS* selects BW, the automatic lock conversion mechanism downgrades BW to NBW during the lock conflict resolution. For a file with 4 stripes, *SeqDLM* outperforms DLM-Lustre by 3.6× and 10.3× when the write size is 47,008 bytes and 47,008 × 16 bytes, respectively. For a file with 8 stripes, the speedup is 2.0× and 6.2×, respectively. The speedup of *SeqDLM* is not linear because writes across stripes decrease the parallelism of lock servers. Fig. 22 shows the total IO time of each test. Similar to the test on a single-striped file, the high performance of *SeqDLM* is derived from the relatively shorter PIO time.

D. Tile-IO Benchmark

Tile-IO is widely used to simulate workloads in visualization and numerical applications [15], [17]. We use it to show the efficiency of *SeqDLM* to support atomic non-contiguous writes. In the test, we use 96 clients to write 8 × 12 tiles. The tiles are stored in a shared file with different stripes. The stripe size is 1 MB. Each tile contains 20,480 × 20,480 pixels with 4 bytes size and there is a 100-pixel horizontal overlap and a 100-pixel vertical overlap between tiles. In total, each client writes 20,480 non-contiguous blocks with the total size of 1.6 GB atomically and the writes among different clients may overlap. We compare *SeqDLM* with DLM-datatype [17]. Using *SeqDLM*, each client requires a lock with a minimum range covering all of the non-contiguous writes for each stripe. For example, for two non-contiguous writes with the ranges [4KB, 8KB] and [16KB, 20KB] in one stripe, we require a lock with the range [4KB, 20KB]. *SeqDLM* incurs more lock conflicts than DLM-datatype. However, because it can decouple data flushing from the lock conflict resolution,

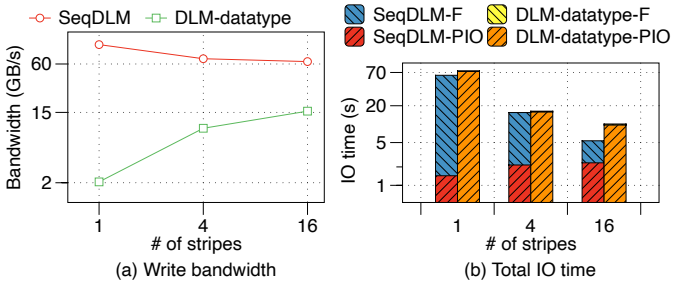

Fig. 23: Tile-IO write bandwidth and total IO time.

SeqDLM outperforms DLM-datatype by 51.0× to 4.1× for the stripe number from 1 to 16 (shown in Fig. 23).

E. VPIC-IO Workload

VPIC-IO [31] is an I/O kernel developed based on a particle physics simulation's I/O pattern, in which processes of the application write a certain number of particles into a HDF5 file with several iterations. Each particle consists of 8 variables and the size of each variable is 4 bytes. We use h5bench [32] to generate the workload. The workload has three phases. (1) It creates and initializes a shared HDF5 file. (2) It writes a certain number of particles into the HDF5 file in parallel with several iterations. In each iteration, each process of the application writes 8 1-D variables and the data of each variable in one iteration are contiguous in the file. When writes from all processes have been done, the data may be still in the caches of PFS. (3) At the end, it flushes the data to disk. Usually, phase (2) is interleaved with computations, so higher IO performance in phase (2) is desired. In the test, we record the time spent on phase (2) as the PIO time and that on phase (3) as the F (flushing) time, similar to evaluations in §V-C and §V-D.

We compare *SeqDLM* with DLM-Lustre in *ccPFS*. We also include the results evaluated on Lustre. The configuration of *ccPFS* and Lustre is the same as that in §V-C. For *ccPFS*, we leverage an IO forwarding software (IOF) that is used in the Sunway Taihulight [33] to provide POSIX-syntax API to transparently support multiple scientific IO libraries. It intercepts system calls and forwards IO operations to its local IO daemon. Each daemon has 8 threads to perform IO on *ccPFS* using the lib*ccPFS* API. To make a fair comparison, we also evaluate Lustre as the back-end of the IOF (Lustre-IOF). We use 16 nodes as data servers and 80 nodes as clients. In the test, 1,280 processes (each client runs 16 processes) concurrently write 10.7×10^9 particles (with the data size of 320 GB) to a shared HDF5 file with 1, 4 and 16 stripes. Each process totally writes 256 MB data. We use the independent IO to perform these writes. To evaluate the effect of write size on write performance, we set the number of particles that each process writes in one iteration to 65,536 (each write size is 256 KB) and 262,144 (each write size is 1 MB). The number of the iterations for the two cases is 128 and 32, respectively.

The write performance in phase (2) is shown in Fig. 24. The write bandwidth in all cases increases as the number of

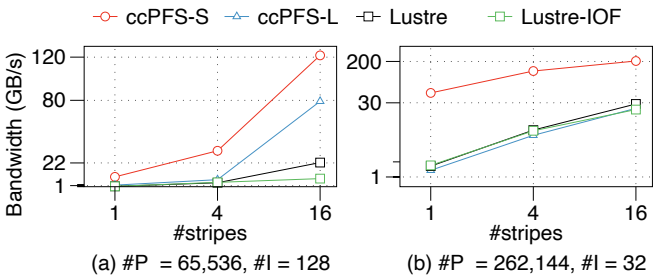

Fig. 24: VPIC-IO write bandwidth. *ccPFS*-S: *ccPFS* using *SeqDLM*, *ccPFS*-L: *ccPFS* using DLM-Lustre, Lustre-IOF: Lustre as the backend of the IOF. P: particles per process in one iteration, I: iterations.

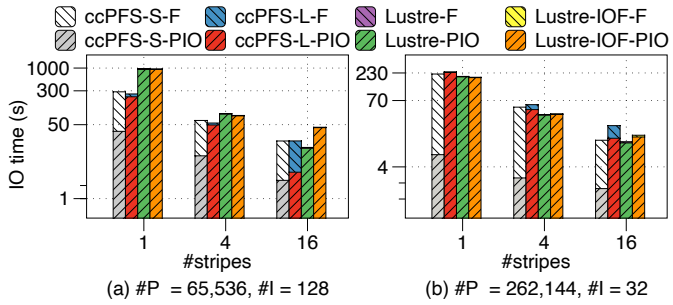

Fig. 25: VPIC-IO PIO time and F time. *ccPFS*-S: *ccPFS* using *SeqDLM*, *ccPFS*-L: *ccPFS* using DLM-Lustre, Lustre-IOF: Lustre as the backend of the IOF. P: particles per process in one iteration, I: iterations.

stripes increases because a file with more stripes not only helps take advantage of the storage devices but also reduces lock contention on each stripe. With 16 stripes and a smaller write size, IOF slightly reduces Lustre performance because with the IOF, IO requests from 16 processes are shipped to 8 threads in the forwarding daemon, which decreases the parallelism. From Fig. 24(a), we see that DLM-Lustre in both *ccPFS* and Lustre gets higher performance with 16 stripes. That is because a smaller write size with a larger number of stripes incurs lower contention due to the optimization of DLM-Lustre for lock contention. The performance of DLM-Lustre in *ccPFS* is better than that in Lustre because the registered memory pool in *ccPFS* is more efficient for IO with smaller write size. Even in this case, *SeqDLM* outperforms DLM-Lustre. In total, in *ccPFS*, *SeqDLM* outperforms DLM-Lustre by 6.2× and 1.5× with the stripe number 1 and 16, respectively. With larger write size, the performance gap in the IO path between *ccPFS* and Lustre is small. Thus, DLM-Lustre in the two systems achieves similar performance, as shown in Fig. 24(b). In this case, *SeqDLM* outperforms DLM-Lustre by 34.8× and 8.8× with the stripe number 1 and 16, respectively.

Fig. 25 shows the total time of phase (2) and phase (3). The higher performance of *SeqDLM* is because it decouples data flushing from lock conflict resolution, resulting in a relatively shorter PIO time. Compared with DLM-Lustre, when using *SeqDLM*, data servers of *ccPFS* need to update the extent cache when handling write requests and execute the extent cache cleaning task asynchronously. From Fig. 25, we see that the total time of *SeqDLM* and DLM-Lustre is similar, which means the extent cache and the cache cleaning mechanism have little impact on the IO performance of data servers.

VI. Related Work

Distributed lock. Lock services, such as Chubby [34], are used as coarse-grained synchronization mechanisms among servers. They are mainly designed for reliability and availability. While our work focuses on distributed locks in PFSes which are designed to synchronize data accesses from multiple clients while guaranteeing distributed client-cache coherence. Several works [12], [17], [35] reduce lock conflicts by limiting lock range expanding and client-cache. In contrast, *SeqDLM* intends to reduce the overhead of lock conflict resolution.

Sequencer mechanism. Many systems decouple ordering from data path to extract parallelism of conflicting writes [36], [37]. A distributed shared log abstract [24], [38], [39] allows parallel appends to run with the speed of a global sequencer. Tran et al. [5] uses versioning to support overlapping MPI-IO. Similarly, *SeqDLM* also uses ordering to allow conflicting data flushing to be completed asynchronously. In contrast, *SeqDLM* allows the sequence to be cached in clients for reuse (in the form of a lock) and ensures all conflicting writes have been flushed to disk before a read starts.

Caching in distributed file systems. GPFS [8], Lustre [7], BeeGFS [1], Ceph [3], and NFS [40] support client data cache. BurstFS [15] uses client-cache to batch bursty metadata writes. Caching introduces the overhead of ensuring coherence. Currently, developers have to make a trade-off between performance and coherence. *ccPFS* leverages *SeqDLM* to guarantee distributed client-cache coherence while achieving high performance both under low contention and high contention.

VII. Conclusion and Future Work

In this paper, we proposed *SeqDLM* to reduce the overhead of the write-write lock conflict resolution while keeping the same semantics as the traditional DLMs. We developed *ccPFS* to evaluate *SeqDLM*. Evaluations show *SeqDLM* achieves high write performance both under high contention and low contention. In the future, we seek to apply *SeqDLM* in Lustre PCC [41] to make it efficiently support parallel writes to shared files. Besides, we also seek to optimize *ccPFS* to be used as a general distributed coherent cache layer for traditional PFSes so that they can efficiently support high contention IO.

Acknowledgment

We sincerely thank the anonymous reviewers for their valuable feedback, which greatly improved this paper. This Work is supported by National Key Research & Development Program of China (2020YFC1522702), Natural Science Foundation of China (62141216, 61877035), and Tsinghua University - Meituan Joint Institute for Digital Life.

REFERENCES

[1] "BeeGFS," https://www.beegfs.io.

[2] "GlusterFS," https://www.gluster.org.

[3] 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*, ser. OSDI '06. USA: USENIX Association, 2006, p. 307–320.

[4] "Tile-IO," https://www.vi4io.org/tools/benchmarks/mpi-tile-io.

[5] V.-T. Tran, B. Nicolae, G. Antoniu, and L. Bougé, "Efficient support for MPI-I/O atomicity based on versioning," in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2011, pp. 514–523.

[6] R. Ross, L. Ward, P. Carns, G. Grider, S. Klasky, Q. Koziol, G. K. Lockwood, K. Mohror, B. Settlemyer, and M. Wolf, "Storage systems and input/output: Organizing, storing, and accessing data for scientific discovery. report for the doe ascr workshop on storage systems and i/o. [full workshop report]," 9 2018.

[7] P. Braam, "The lustre storage architecture," *arXiv preprint arXiv:1903.01955*, 2019.

[8] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk file system for large computing clusters," in *Conference on File and Storage Technologies (FAST 02)*, 2002.

[9] K. Thomas, "Programming locking applications," IBM Corporation, Tech. Rep., 2001.

[10] P. J. Braam, "Scalable locking and recovery for network file systems," in *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, ser. PDSW '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 17–20.

[11] 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 Conference on High Performance Computing Networking, Storage and Analysis*. IEEE, 2009, pp. 1–12.

[12] M. Moore, P. Farrell, and B. Cernohous, "Lustre lockahead: Early experience and performance using optimized locking," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 1, p. e4332, 2018.

[13] B. Yang, X. Ji, X. Ma, X. Wang, T. Zhang, X. Zhu, N. El-Sayed, H. Lan, Y. Yang, J. Zhai *et al.*, "End-to-end {I/O} monitoring on a leading supercomputer," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 379–394.

[14] M. Vef, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS - A temporary burst buffer file system for HPC applications," *J. Comput. Sci. Technol.*, vol. 35, no. 1, pp. 72–91, 2020.

[15] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 807–818.

[16] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and friends: A proposal for an exascale storage system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. IEEE Press, 2016.

[17] A. Ching, W.-k. Liao, A. Choudhary, R. Ross, and L. Ward, "Noncontiguous locking techniques for parallel file systems," in *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 2007, pp. 1–12.

[18] W. keng Liao, K. Coloma, A. Choudhary, L. Ward, E. Russell, and N. Pundit, "Scalable design and implementations for MPI parallel overlapping I/O," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 11, pp. 1264–1276, 2006.

[19] T. Patel, S. Byna, G. K. Lockwood, and D. Tiwari, "Revisiting I/O behavior in large-scale storage systems: The expected and the unexpected," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[20] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, p. 375–408, Sep. 2002.

[21] P. Sigdel, X. Yuan, and N.-F. Tzeng, "Realizing best checkpointing control in computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 2, pp. 315–329, 2021.

[22] "IOR," https://github.com/hpc/ior.

[23] "fakeWrite," https://jira.whamcloud.com/browse/LU-7655.

[24] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber, "CORFU: A distributed shared log," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, Dec. 2013.

[25] M. Vilayannur, P. Nath, and A. Sivasubramaniam, "Providing tunable consistency for a parallel file store," in *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, ser. FAST'05. USA: USENIX Association, 2005, p. 2.

[26] F. Wang, S. Oral, G. Shipman, O. Drokin, T. Wang, and I. Huang, "Understanding lustre filesystem internals," OakRidge National Laboratory, Tech. Rep., 2009.

[27] Intel, "Collective and RPC Transport (CaRT)," https://github.com/daos-stack/daos/tree/master/src/cart.

[28] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, 2013, pp. 1–8.

[29] "IO500," https://io500.org.

[30] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. IEEE Press, 2008.

[31] K. Wu, S. Byna, and B. Dong, "VPIC IO utilities," 12 2018.

[32] T. Li, S. Byna, Q. Koziol, H. Tang, J. L. Bez, and Q. Kang, "h5bench: HDF5 I/O Kernel Suite for Exercising HPC I/O Patterns," in *Proceedings of Cray User Group Meeting, CUG 2021*, 2021.

[33] Q. Chen, K. Chen, Z.-N. Chen, W. Xue, X. Ji, and B. Yang, "Lessons learned from optimizing the sunway storage system for higher application i/o performance," *J. Comput. Sci. Technol.*, vol. 35, no. 1, p. 47–60, jan 2020.

[34] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. USA: USENIX Association, 2006, p. 335–350.

[35] C. Gray and D. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, ser. SOSP '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 202–210.

[36] X. Liao, Y. Lu, E. Xu, and J. Shu, "Write dependency disentanglement with HORAE," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 549–565.

[37] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho, "Barrier-enabled IO stack for flash storage," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 211–226.

[38] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed data structures over a shared log," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 325–340.

[39] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. Malkhi, "vCorfu: A cloud-scale object store on a shared log," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 35–49.

[40] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS version 3: Design and implementation." in *USENIX Summer*. Boston, MA, 1994, pp. 137–152.

[41] Y. Qian, X. Li, S. Ihara, A. Dilger, C. Thomaz, S. Wang, W. Cheng, C. Li, L. Zeng, F. Wang, D. Feng, T. Süß, and A. Brinkmann, "LPCC: Hierarchical persistent client caching for lustre," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We run experiments to demonstrate the correctness and effectiveness of the SeqDLM design using micro benchmarks and typical scientific workloads. The scientific workloads are generated using IOR benchmark, app-tile-IO benchmark and h5bench benchmark. In the experiments, we use NFS to provide the namespace service of ccPFS. When accessing a file, ccPFS first creates/opens a file on NFS. After that, it takes the inode number of the file as the FID (file identifier) and distributes the stripes and the lock resources among servers using the FID. The experiments were performed on a 96-node cluster. They can be run using the scripts in our artifact. Refer to the file install/ccpfs/ad/README.md in the artifact for details.

The hardware details of the cluster are described in the experimental setup section in our paper. The software details are as follows.

Operating systems and versions: CentOS version 7.7 running Linux kernel 3.10.0-862.14.4.el7.

Software and versions: Most of the software needed by ccPFS are installed from the standard repository of CentOS version 7.7. We provide a script in our artifact to install them. Here, we list the software that are not included in the standard repository. These software are included in our artifact.

(1) CaRT from DAOS (https://github.com/daos-stack/daos) with commit ID d4a7ae714f45da8828cd523b08039f656510a05d.

(2) Mercury (https://github.com/mercury-hpc/mercury) with commit ID 932ad534d8e3f8106e85d0931c20ee13f0e42188.

(3) libfabric (https://github.com/ofiwg/libfabric) with commit ID 5b0a7b2b516d20a3896e26381b8951e64c4824a5.

(4) IOR (https://github.com/hpc/ior) with commit ID 657ff8ad8f136b14e57677dbe0175e8e875346d5.

(5) mpi-tile-IO (https://www.mcs.anl.gov/research/projects/pio-benchmark/code/mpi-tile-io-01022003.tgz)

(6) h5bench (https://github.com/hpc-io/h5bench) with commit ID 81bd930510453ebe28b324daa329414e44f88e21

We modify the code of IOR and mpi-tile-IO to use ccPFS API to perform IO. We modify h5bench to record the time of different phases. These patches are included in our artifact.

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://doi.org/10.5281/zenodo.6985090`

Artifact name: ccPFS Prototype

Reproduction of the artifact without container: ccPFS uses IB verbs to transfer data between clients and servers and stores data in NVMe SSDs. We have no experience on deploying and testing large scale of a docker cluster which supports both NVMe SSD and Inifiniband network. Thus, we provide our artifact in a package (ccpfs-bin.tar.gz) that is already built and tested on nodes using CentOS 7.x. We provide a script to install the software dependencies that exist in the standard repository of CentOS version 7.x For others, we include them in the provided package.

Our artifact can be reproduced on nodes with CentOS version 7.x with the following steps.

1. Extract ccpfs-bin.tar.gz to some directory. In the following, we take the directory "/opt" as an example. The command to extract ccpfs-bin.tar.gz is "tar -zxvf ccpfs-bin.tar.gz -C /opt".

2. Export "/opt/ccpfs" as a writeable NFS shared directory by adding "/opt/ccpfs *(rw,no_root_squash)" to file "/etc/exports" and restarting the NFS service. If the NFS software are not installed, install it using the command "yum install -y nfs-utils".

3. Create a directory "/home/ccpfs" on each node that will run the evaluations, and mount the shared directory into "/home/ccpfs" with the command "mount NFSSERVER_IP:/opt/ccpfs /home/ccpfs". The mount point "/home/ccpfs" cannot be changed because we include the software dependencies that have been built in ccpfs-bin.tar.gz and use the absolute path with the prefix "/home/ccpfs/install" to avoid software version conflicts.

4. Install the other necessary software dependencies that are available in the standard CentOS repositories. This step should be run on all of the nodes that run the evaluations. Users can install the software dependencies with the command "sh /home/ccpfs/install-deps.sh". Alternatively, users can also directly run the following command: "yum install -y hwloc hwloc-devel libyaml gcc libatomic mpich-3.2 mpich-3.2-autoload mpich-3.2-devel git gcc-c++ libtool boost-devel cmake libuuid-devel openssl-devel libyaml-devel libibverbs".

5. In the evaluations, we use a shared directory "/home/ccpfs/install/ccpfs/ccpfs" on NFS to provide namespace service for ccPFS. Users need to use the following command to create the shared directory: "mkdir /home/ccpfs/install/ccpfs/ccpfs".

6. Assign a directory to store local data for each server. For example, if we use /mnt/nvme1n1/data/data/ to store local data, we execute the following command: "ln -s /mnt/nvme1n1/data/data /home/ccpfs/install/ccpfs/data".

7. Enter the directory "/home/ccpfs/install/ccpfs/ad/" and record hostnames or IPs of nodes that will run the evaluations in the file "hosts", which is needed by mpirun. We need 96 nodes to perform all of the evaluations. We also provide two scripts that can be run on a single node for functional test. Referring to the file "/home/ccpfs/install/ccpfs/ad/README.md" for details.

8. Enter the directory "/home/ccpfs/install/ccpfs/ad/" and run the scripts in the directory to perform the evaluations. Referring to the file "README.md" in the directory for details.