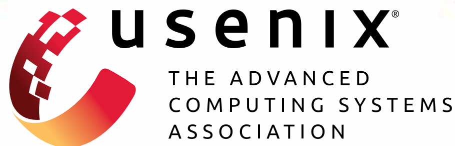


Combining Buffered I/O and Direct I/O in Distributed File Systems

Yingjin Qian, *Data Direct Networks;* Marc-André Vef, *Johannes Gutenberg University Mainz;* Patrick Farrell and Andreas Dilger, *Whamcloud Inc.;* Xi Li and Shuichi Ihara, *Data Direct Networks;* Yinjin Fu, *Sun Yat-Sen University;* Wei Xue, *Tsinghua University and Qinghai University;* André Brinkmann, *Johannes Gutenberg University Mainz*

https://www.usenix.org/conference/fast24/presentation/qian

This paper is included in the Proceedings of the 22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

Open access to the Proceedings of the 22nd USENIX Conference on File and Storage Technologies is sponsored by

NetApp®

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

Combining Buffered I/O and Direct I/O in Distributed File Systems

Yingjin Qian
Data Direct Networks

Marc-André Vef
Johannes Gutenberg University Mainz

Patrick Farrell
Whamcloud Inc.

Andreas Dilger
Whamcloud Inc.

Xi Li
Data Direct Networks

Shuichi Ihara
Data Direct Networks

Yinjin Fu
Sun Yat-Sen University

Wei Xue
Tsinghua University & Qinghai University

André Brinkmann
Johannes Gutenberg University Mainz

Abstract

Direct I/O allows I/O requests to bypass the Linux page cache and was introduced over 20 years ago as an alternative to the default buffered I/O mode. However, high-performance computing (HPC) applications still mostly rely on buffered I/O, even if direct I/O could perform better in a given situation. This is because users tend to use the I/O mode they are most familiar with. Moreover, with complex distributed file systems and applications, it is often unclear which I/O mode to use.

In this paper, we show under which conditions both I/O modes are beneficial and present a new transparent approach that dynamically switches to each I/O mode within the file system. Its decision is based not only on the I/O size but also on file lock contention and memory constraints. We exemplary implemented our design into the Lustre client and server and extended it with additional features, e.g., delayed allocation. Under various conditions and real-world workloads, our approach achieved up to 3× higher throughput than the original Lustre and outperformed other distributed file systems that include varying degrees of direct I/O support by up to 13×.

1 Introduction

High-performance computing (HPC) clusters traditionally store data on parallel file systems [4, 9, 14, 15, 49, 57]. They export local file or object storage from a collection of server nodes to clients, allowing applications on a client to access files on remote servers as if they were stored locally. Existing applications constantly scale to higher core counts and proportionally increase their I/O volume. New HPC applications from machine learning and AI are creating new access patterns that challenge previous optimizations for parallel file systems by increasing random accesses and heavy metadata traffic. As a result, I/O is increasingly becoming a performance bottleneck for many scientific applications.

File systems typically cache data and metadata in main memory to reduce the number of required I/Os to the storage backend. For example, Linux's default I/O mode is *buffered I/O* where the kernel caches read and write operations in the Linux page cache to help optimize I/O submitted to storage. Almost all standard applications running on a single server can benefit from page caching during I/O. Buffered I/O also improves the performance of many HPC applications that run on large clusters and store data on parallel file systems.

An alternative to buffered I/O is *direct I/O*. Files opened with the `O_DIRECT` flag bypass the caching layer in the kernel and send I/Os directly to the storage system. This is particularly useful when an application itself buffers read and write operations, avoiding a "double buffer" situation, such as in databases. To use direct I/O, an application must meet certain alignment criteria. The alignment constraints are usually determined by the disk driver, the disk controller, and the system memory management hardware and software. This requirement severely limits the use of direct I/O by applications.

Intuitively, buffered I/O should perform better than direct I/O because the read-ahead and write-back optimizations of buffered I/O can bring the performance of buffered I/O mode close to the level of memory access. Using direct I/O therefore typically results in a prolonged process. However, this paper shows that this is not always the case.

The reason is that caching data in the kernel page cache is not free, especially if the cached data has poor reuse characteristics. First, buffered I/O induces additional copy operations to move data between the kernel cache and the application. Second, the overhead of interacting with the kernel page cache and page management is considerable. Moreover, when memory becomes scarce, page reclamation must free old pages to allocate memory for the current I/O operation. The resulting cache thrashing can then significantly degrade performance.

An additional cost of buffered I/O in parallel file systems is the cost of managing complex distributed range locks to support client-side caching with strong consistency. If the file system locks only the necessary (small) portions of a concurrently accessed file, it requires many *remote procedure calls* (RPCs) between clients and the lock manager, while using larger expanding locks may lead to false lock contention between clients and many lock revocation messages [39].

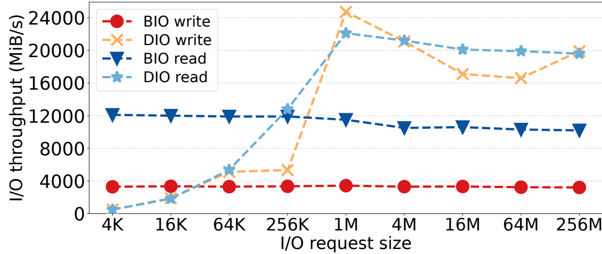


Figure 1: Local `ldiskfs` performance with various I/O sizes for buffered I/O (BIO) and direct I/O (DIO).

The primary benefit of direct I/O is to reduce CPU utilization for file reads and writes by eliminating the copy from the cache to the user buffer and minimizing the number of lock revocations. Therefore, the advantages and drawbacks of the two I/O modes complement each other. Buffered I/O simplifies programming and can yield performance benefits in many situations. However, for sequential I/O to very large files, direct I/O with large transfer sizes can provide the same or better performance as buffered I/O with much less CPU overhead and memory usage. In addition, direct I/O can also improve performance for small writes when many nodes with interleaved file offsets concurrently modify a file.

Figure 1 shows an example of this tradeoff when 16 threads run the `fio` benchmark on a local Lustre `ldiskfs` device. Each thread used separate files and wrote and read 20 GiB of data for I/O sizes between 4 KiB and 256 MiB. The write aggregation and read-ahead optimizations of buffered I/O resulted in a stable write performance of 3 GiB/s and a read performance of 11 GiB/s, almost independent of access sizes. This *performance wall* for buffered I/O depends on available memory for caching and page cache overhead and is independent of available storage bandwidth or number of the attached storage system. We also experienced the same behavior for other file systems like BeeGFS [25] or NFS [22].

The performance of direct I/O for small I/O sizes in Figure 1 is significantly lower than in buffered I/O mode. In this case, direct I/O suffers from latencies induced by synchronous writes to the storage backend. For bigger I/O sizes however, direct I/O benefits from not performing unnecessary copy operations and not having to manage the page cache, reaching a performance that exploits the potential of the backend SSDs.

To the best of our knowledge, we are the first to evaluate combining buffered I/O and direct I/O in the parallel file system itself. Based on our empirical results, we designed and implemented a new I/O path engine for the Lustre parallel file system that can automatically switch between buffered I/O and direct I/O modes. In contrast to previous work on BeeGFS [5], this switch is not only based on the size of requests but also considers memory pressure on compute nodes and lock contention on files. We also introduce a mechanism that supports adaptive switching between buffered I/O and direct I/O on storage servers.

We compare this new architecture with BeeGFS and OrangeFS [2]. We chose these two file systems as BeeGFS can switch between buffered I/O and direct I/O based on a fixed threshold [5], whereas OrangeFS in the tested implementation only performs direct I/O [17] on the client side. Our evaluation uses a variety of workloads, including microbenchmarks, macrobenchmarks, and real-world HPC workloads.

We show that our approach can effectively combine buffered I/O and direct I/O, selecting the best-performing I/O mode for a given I/O size and system state. Compared with the original Lustre version, our approach achieved up to 3× higher throughput for real-world workloads (that use many heterogeneous I/O sizes) and outperformed BeeGFS and OrangeFS by up to 13× and up to 10×, respectively. Moreover, we present the I/O statistics in which the I/O mode switch was triggered.

The remainder of this paper is organized as follows. First, Section 2 discusses the necessary background and motivates our work. Section 3 presents our new Lustre I/O engine. Next, Section 4 evaluates different parameter settings and compares our approach with BeeGFS and OrangeFS. Section 5 discusses related work, and we conclude with Section 6.

2 Background and motivation

In this section, we present a detailed comparative analysis between buffered I/O and direct I/O and then introduce the performance impact of page caching and I/O lock contention on buffered I/O to motivate our design for higher I/O performance in HPC systems.

2.1 Buffered I/O vs. direct I/O

Linux and most other operating systems offer buffered and direct I/O modes for file access. In the buffered I/O mode, the virtual file system first buffers all read and write requests in the kernel page cache. This is the default file access mode and is easy to integrate into applications as they do not need to deal with I/O size and alignment constraints. A major advantage of buffered I/O is that it can hide the latency of storage accesses when data is accessed more than once. The direct I/O mode in contrast transfers data directly between the application and the storage device without a data copy, but the data must meet specific size and alignment constraints.

Table 1 provides a high-level comparison of the two I/O modes. A key advantage of buffered I/O is its ability to prefetch data using read-ahead and to aggregate small writes using write-back caching. In both cases, small I/O requests from the application can be transformed into large I/O operations to the underlying storage system. Asynchronous write-back caching and read-ahead are perfect for hiding the latency of slow storage devices, such as spinning disks, and can perform close to the speed of the `memcpy()` operation. Buffered

Table 1: Comparison between two I/O modes.

I/O case	Buffered I/O	Direct I/O
Small I/O size	✓	X
High latency storage	✓	X
Unaligned I/O	✓	X
Large, sequential I/O	X	✓
Many running processes/nodes	X	✓
System under memory pressure	X	✓

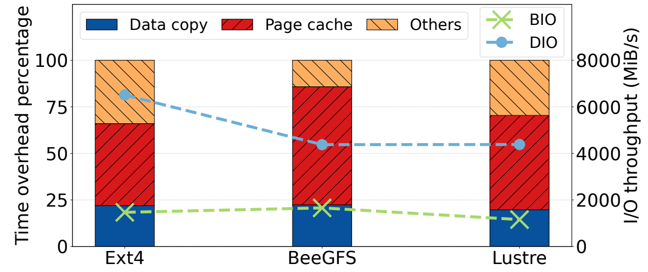


Figure 2: I/O time breakdown for buffered I/O writes.

I/O also has no requirements on the read and write size and alignment and is therefore convenient to use by programmers.

A major drawback of buffered I/O is its poor single-stream performance when data cannot be reused multiple times. It also creates many lock conflicts when multiple processes from different nodes write to a shared file on a networked file system with strong consistency guarantees. There is also contention within a single node's page cache when multiple cores are writing to a single file.

Direct I/O does not use page caching and transfers data directly between application memory and the storage device. It can provide near-device performance for large I/O sizes and does not slow down when scaling the number of processes and nodes. It can also reduce memory pressure because data is not prefetched or cached. The downside is that direct I/O cannot hide the latency for slow devices or small I/Os. Also, direct I/O requires that the I/O size and the offset in memory are aligned with the page size, so most applications must be significantly modified to use this I/O mode.

2.2 Impact of page caching and data copies on buffered I/O

The page cache used by buffered I/O induces additional copies between user space and the page cache. Its management requires, e.g., page allocation, locking, and LRU list management for aging and reclaiming. We therefore designed an experiment to measure the page cache overhead for sequential write operations from a single thread for a total I/O size of 2,560 GiB. We used the IOR benchmark [1] and the `perf` [28] profiling utility to collect and analyze the corresponding performance and trace data. Figure 2 shows the results for the local file system Ext4 and the network file systems Lustre and BeeGFS. These file systems spent about 20% of their time on copying data between the application and the page cache and more than 40% on page cache management. The figure also shows the resulting buffered I/O performance and the performance of direct I/O in the same setting.

This page cache overhead was also observed in previous work. For example, Corbet [16] describes that even (seemingly harmless) reads that span a dataset larger than the memory size can lead to page cache thrashing and huge performance drops once the page cache is full (see also [35, 44]). It is therefore interesting to understand why page cache management is so costly, even for such simple use cases.

Page cache management includes page allocation and page reclaim, among others. The page allocation process allocates clean pages for newly accessed data and adds the pages to the page cache. During memory pressure, Linux must reclaim some of the previously allocated pages from the cache to make room for newly allocated pages. In our example, reclaiming pages requires not only evicting pages from the page cache but also writing the data back to the storage system (see also [21]). A closer look at the profiling data shows that IOR running on BeeGFS, for example, spent more than 42% of its time in the `pagecache_get_page()` function. The reason is that `pagecache_get_page()` can only return under memory pressure after it has received a clean page. These clean pages must first be generated by the `kswapd` daemon. `kswapd` therefore constantly launched many `kworker` background threads that asynchronously wrote back these dirty pages using the native BeeGFS write functions, which in turn can impose an overhead and can lead to page cache thrashing. An additional page management overhead of 20% attributed to setting pages dirty by calling `__set_page_dirty_nobuffers()`.

Direct I/O does not interact with the page cache and can perform large sequential writes directly to the backend storage. The performance comparison in Figure 2 therefore shows that direct I/O on the local file system Ext4 can increase performance by nearly five times, while Lustre and BeeGFS have smaller gains because they require additional bulk data transfers over the network.

2.3 I/O locking and contention in Lustre

The Lustre file system stores data on object storage targets (OSTs) exporting local disk file systems through object storage servers (OSSs). Similarly, metadata is stored on metadata targets (MDTs) which are accessed through metadata servers (MDSs). Both data and metadata performance and capacity can be scaled by including more servers [9]. Lustre clients run on the compute nodes and access the storage and metadata servers through a high-speed network. Lustre supports client-side caching of data and metadata to reduce the impact of network round-trip times [45, 46]. It uses a *distributed lock*

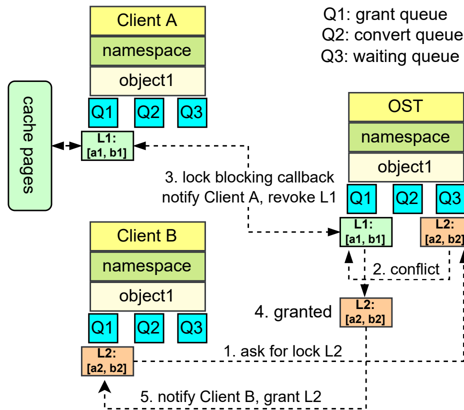


Figure 3: Lock blocking callback under write-back.

manager (DLM) [31, 37, 49] to protect the cached data and metadata from concurrent accesses by other clients. Lustre manages DLM locks in namespaces, where each Lustre OST or MDT has a separate lock namespace for its local objects.

Files are protected by read and write byte-range locks, allowing multiple clients to access or modify different parts of a shared file. A client requests a lock covering exactly the required I/O range (aligned with the page cache). The server attempts to optimize this request by expanding the lock to the largest non-conflicting range. The locks cached by a client are not released immediately and are instead revoked asynchronously through a callback in case of a lock conflict due to age or when exceeding cache sizes. Based on the locality principle, this can reduce lock traffic between clients and servers and improve performance, especially when only a single client accesses a file. Nevertheless, it can also result in heavy lock contention and false lock sharing for concurrent writes on a shared file from multiple clients [39].

Figure 3 shows the required steps to allocate a lock for write-back caching data in the range `[a2,b2]` by client B. We assume that client A previously wrote data to the same file and still keeps a lock `L1=<[a1, b1]>` in its local lock namespace. Now, client B requests a lock `L2=<[a2, b2]>` on the same object. The server detects that `L2` conflicts with `L1` because the two lock ranges `[a2, b2]` and `[a1, b1]` intersect and notifies client A to revoke `L1` via the lock-blocking callback. Client A then flushes the dirty pages to the server, clears the client cache, and releases `L1`. Afterward, the server grants `L2` to client B, and then client B can write to the file.

Lock conflict resolution and possible lock ping-pong are expensive processes and can significantly reduce I/O performance when writing to shared files. Therefore, we present a mechanism for using direct I/O with server-side locking to eliminate the lock callbacks for conflict I/Os next.

3 Design and implementation

Our design consists of four main components. Its switching algorithm automatically selects the I/O mode on both the client and the server to match request sizes and access patterns. Server-side adaptive locking reduces lock congestion on shared files when the I/O pattern has no access locality or when many clients access a file in parallel. Server-side delayed allocation improves strided I/O performance, and support for unaligned direct I/O accesses simplifies programmability. We have implemented our approach in the Lustre parallel file system. However, the general idea is applicable to other distributed file systems that also use a DLM.

3.1 Combining buffered I/O and direct I/O

We introduce a fully transparent hybrid I/O path engine that automatically switches between buffered I/O and direct I/O in the Lustre client and server.

***autoIO* – transparent direct I/O in the client:** *autoIO* uses the I/O request size, lock contention, memory pressure, and access locality to decide whether a client's I/O request should be handled as buffered or direct I/O. Algorithm 1 shows the corresponding decision tree for autoIO to automatically switch between buffered I/O and direct I/O.

If an I/O request is smaller than the *small I/O threshold*, autoIO uses buffered I/O. If the I/O request size is larger or equal to the *large I/O threshold*, autoIO uses direct I/O. Between both thresholds, autoIO uses buffered I/O by default and first checks whether the file is under lock contention due to conflicting accesses from multiple clients, and if so, switches to direct I/O. This is advantageous in combination with our adaptive server-side locking, which is discussed later in this section. Next, autoIO considers the client's current memory pressure and cache reuse. AutoIO therefore limits the number of cached pages of a file to 1 GiB (default) and switches to direct I/O when the cached pages or a corresponding cgroup reach 95% of their allowed limit. Also, if the I/O workload

Algorithm 1 The autoIO decision algorithm

1: **function** DECIDE_IO_MODE(file, io_size, cfg)
2: **if** $(io_size \geq cfg.large_io_threshold)$ **then**
3: return DIO;
4: **else if** $(io_size < cfg.small_io_threshold)$ **then**
5: return BIO;
6: **else if** file_is_under_lock_contention(file) **then**
7: return DIO;
8: **else if** client_is_under_memory_pressure(file, cfg) **then**
9: return DIO;
10: **else if** file_lacks_access_locality(file, io_size, cfg) **then**
11: return DIO;
12: **else**
13: return BIO;

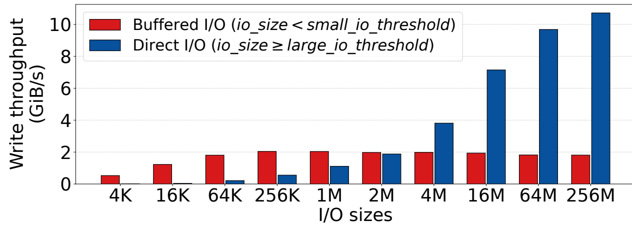


Figure 4: I/O streaming throughput for autoIO with various small and large I/O thresholds.

lacks access locality and the cached pages are not reused, autoIO switches to direct I/O for subsequent I/O accesses larger than the small I/O threshold.

For lock contention detection, autoIO leverages an already existing detection mechanism in Lustre. The other default parameters are based on preliminary experiments and are configurable by the user.

We ran I/O streaming experiments on a Lustre system with eight storage servers to determine the initial small and large I/O thresholds (see Section 4 for the experimental setup). We used IOR [1] with various I/O request sizes ranging from 4 KiB to 256 MiB. Figure 4 presents the results for each I/O size for two cases: First, the small I/O threshold was set larger than the I/O request size, resulting in buffered I/O. Second, the large I/O threshold was set smaller than the I/O request size, resulting in direct I/O. Based on the results, we set the large I/O threshold to 2 MiB by default. We further set the small I/O threshold to 32 KiB. Therefore, if Lustre detects lock contention, memory pressure, or a lack of access locality, autoIO switches to direct I/O in the range [32 KiB, 2 MiB). Section 4 provides additional I/O statistics when a specific case was triggered and shows the performance benefits of switching to direct I/O across workloads within this range.

Note that autoIO does not switch to buffered I/O when a user opens a file with the `O_DIRECT` flag.

Adaptive server-side locking for direct I/O Bulk-synchronous applications [10] represent some of the most dominant workloads in HPC, where applications typically alternate compute and I/O phases at step boundaries. Often, such workloads access a single shared file at a step boundary, where each process accesses its own non-conflicting region. Examples include Nek5000 [20] for computational fluid dynamics, VPIC for large-scale plasma physics simulations [8, 12], and checkpointing to a single shared file [7].

This workload type is so common that it is included in the IO500 [32] *ior-hard-write* benchmark where each MPI rank concurrently writes into a single shared file with an I/O size of 47,008 bytes in strided I/O mode. Although the I/O regions do not overlap, this can cause significant lock contention. The reason is that clients must obtain page-aligned extent locks before performing their non-page-aligned I/Os. This can result in many ping-pong lock callbacks that negatively affect I/O throughput.

For direct I/O, our implementation leverages an existing server-side locking mechanism in which clients send their I/O requests directly to the server, which acquires the DLM lock on the client's behalf. Compared to the client-side extent lock, the server-side extent lock has a much lower latency because the server can take the lock before proceeding with the bulk transfer (and free it directly after), saving lock round-trip traffic. Therefore, when a file is under lock contention, the benefits of direct I/O shift towards smaller I/O sizes.

For autoIO, this means that, according to our experiments, it is beneficial to use direct I/O already below the large I/O size threshold but above the small I/O size threshold (see Algorithm 1 line 6) when a file is under lock contention. To determine whether a lock resource is under contention and to what degree, our detection algorithm uses a sliding window counter [59]. If at least 16 (by default) conflicting lock requests are seen over a sliding window of 4 seconds (by default), the file object is considered under contention, which the server reports to the client via the reply. Overall, this can increase the efficiency for I/O requests that fall between the small and large I/O thresholds, as shown later in Section 4.

Server-side adaptive write-back and write-through Lustre servers implement a thread pool model for incoming requests from many clients in parallel. Specifically, Lustre uses an out-of-band data transfer mode, combined with *remote direct memory access* (RDMA) network transfers, to minimize CPU and memory utilization while providing high throughput and scalability. For large incoming I/O requests, each I/O service thread uses a pre-allocated buffer (between 4 MiB and the maximum bulk RPC size) for bulk data transfers to avoid the overhead of the kernel page cache. Depending on the server load, the number of I/O threads can vary from 2 to 512 threads. Therefore, the total memory requirement can be several GiB (RPC size times 512 threads). By default, all I/O requests are immediately submitted to storage by the I/O service thread to avoid memory contention (*write-through*).

Although this works well for large I/O requests, especially when using large bulk RPCs (up to 64 MiB in Lustre), the write-through mode does not fully utilize the available disk bandwidth for small I/O requests (see Figure 1). This is particularly noticeable for latency-sensitive I/O requests, such as when many small files are written and read.

We added an adaptive write-back cache mode to improve I/O performance on the server for such latency-sensitive use cases. Similar to how the Lustre client can switch to direct I/O for non-`O_DIRECT` requests, the Lustre server can switch to buffered I/O mode (*write-back*). According to the results in Figure 1, all I/O requests smaller than 64 KiB are processed in write-back mode using the page cache. For larger I/O requests, Lustre uses its default write-through mode via direct I/O. This allows the server to use the available memory to cache small

I/Os while not overwhelming the cache with large I/Os.

Moreover, we do not implement server-side read-ahead. The reason is that 1. the server memory is a limited resource shared by all clients, and read-ahead data may overwhelm the cache space on a server; and 2. client-driven read-ahead in Lustre is a more efficient way and has already achieved very good performance in most cases.

Note that when using write-back, limited data availability after a crash is a challenge, with the potential to lose non-persisted data. However, this is also the case with buffered I/O in general. Here, `fsync()` is essential to ensure data consistency and durability within the file system. This is in accordance with POSIX which states that a successful `write()` does not guarantee that the data has been committed to disk unless `fsync()` is called. Interestingly, this applies also to direct I/O. Although it bypasses the client's kernel cache, a storage system may or may not apply `O_DIRECT` to other layers of the I/O stack where caching is used. Therefore, it is important to use `fsync()` to flush all cached data to disk.

Overall, our work does not affect metadata durability and consistency as the file system can always recover to a consistent state after a crash. For example, Lustre's recovery mechanism can resend (replay) uncommitted I/O operations on clients to the server once the server resumes operations [43].

Cross-file batching for buffered writes For single files, Lustre clients accumulate an application's dirty pages and asynchronously send them as large bulk RPCs (1 MiB) to the storage servers. This method avoids many small RPCs, and it is therefore more network and disk-efficient. For many small files, however, there may not be enough dirty pages to accommodate a full bulk RPC, delaying the write-back operation. Moreover, I/O RPCs sending dirty pages to the storage servers are restricted to a single file and thus many small RPCs are sent.

Cross-file batching for buffered writes is an optimization strategy for such use cases that involve many small writes across many small files. It batches dirty pages of multiple files into one large bulk RPC, improving network efficiency. The configurable threshold `small_write_threshold` (default 64 KiB) allows Lustre to distinguish whether a file is small enough to benefit from this optimization. Essentially, Lustre clients maintain a small-file list that contains files below the threshold. Once enough dirty pages are placed in the list, batched I/O bulk RPCs are sent to the storage servers.

Consistency challenges Whenever our algorithm triggers transparent and dynamic switching between buffered I/O and direct I/O, there is the potential for data regions from the two I/O modes to overlap. For instance, on the Lustre client, a file region could be written via direct I/O while parts have not been flushed yet and remain in page cache as they were part of a prior buffered I/O operation. If not handled properly, this can cause a consistency conflict. A similar situation could occur on the server with the above-described write-back cache and the default direct I/O path.

On the client, overlapping regions are detected, and dirty pages in that region are flushed first before direct I/O is performed. On the server side, with write-back enabled, dirty pages are not flushed and are reused instead as part of the direct I/O operation. Thus, because clients must flush their cache and servers merge overlapping data, our approach does not change Lustre's strong consistency guarantees. Similar arguments hold for aligning unaligned direct I/O, which we discuss next.

3.2 Unaligned direct I/O

One of the challenges of using direct I/O is that the `O_DIRECT` open flag typically imposes alignment constraints on the length and address of user space buffers and the file offset of I/Os, where the I/O size and offset must be a multiple of 512 bytes and the memory buffer address must also be aligned to 512 bytes [26]. In this context, the 512-byte boundary refers to the logical block size of the underlying storage device, although modern devices use a sector size of 4096 bytes or more [36]. If not all conditions are true, direct I/O is not supported and the error code `EINVAL` is returned.

In general, I/O alignment provides several benefits. For example, it can resolve conflicting *read-modify-write* (RMW) operations on the same block. However, not all applications can align their I/O. This is especially true for applications with a complex I/O stack that is not under the control of the application. Therefore, to maximize the benefits of direct I/O, the underlying file system should handle misaligned I/O in the kernel. We implemented a buffering scheme in the Lustre client to address this challenge. When the user buffer is misaligned, Lustre creates an aligned buffer in the kernel by remotely reading data outside the user buffer up to the alignment boundary. Direct I/O then uses this aligned buffer.

Nevertheless, because aligning a user buffer potentially requires additional memory allocation and data copying, it may be less efficient than using an existing user buffer. However, our analysis showed that allocating and copying to an aligned buffer in the kernel still outperforms buffered I/O, especially for large I/O sizes. This is because buffered I/O spends less than 20% of its time allocating a buffer and copying data to that buffer, while the remaining time is spent locking page caches and managing the kernel cache.

3.3 Efficient RAID I/O via delayed allocation

When Lustre receives I/O write requests for a small file, blocks are allocated at write time, even if the data is written in write-back mode. This strategy can lead to severe file fragmentation for strided I/O (discussed above) when multiple clients are writing data to a single file and are therefore constantly allocating blocks simultaneously. Since magnetic disks are still

used as the primary backend for storing data in parallel file systems, we try to reduce file fragmentation by delaying block allocation on the backend storage.

A common technique to mitigate such file fragmentation is to use *delayed (block) allocation* by deferring data block allocation until the last possible moment before data is flushed to disk. Delayed block allocation is featured in many modern file systems, e.g., EXT4 [13], XFS [29], or BtrFS [48] to reduce fragmentation [51].

Therefore, we have enabled delayed allocation in write-back mode on the server to collect and merge small or non-contiguous I/O requests into large, contiguous I/O requests. This can reduce head thrashing on magnetic disks. It can also reduce the number of RMWs on RAID systems by consolidating full stripes before flushing the data. Small fragmented I/O, on the other hand, immediately writes the data to the RAID controller cache before flushing it in an RMW fashion, significantly impacting I/O performance.

Delayed allocation uses the kernel's write-back mechanism to flush dirty pages and allocate blocks at flush time. By default, Linux's periodic flushing interval is five seconds, during which the disk bandwidth may be underutilized. To flush data continuously, we leverage *extent status trees* (available in Ext4 and ldiskfs) – a data structure to track the status of delayed allocation extents. When a server receives a write request from a client, it allocates an in-memory delayed extent and inserts it into the delayed extent status tree. During insertion, the server looks for other delayed extents to form a contiguous extent. If Lustre detects that a merged extent can form a full extent write, e.g., offset and length are both 1 MiB aligned, the dirty pages from this extent are flushed by a worker thread. In summary, this allows larger continuous I/O buffers to be flushed to the underlying RAID disk system outside of the periodic flushing interval, reducing RMW operations caused by otherwise small extents and improving the overall I/O performance in the process.

4 Evaluation

This section evaluates the performance and benefits of our work under various workloads, including microbenchmarks, macrobenchmarks, and real application workloads. Most of these experiments are compared to BeeGFS and OrangeFS. We chose these two distributed file systems for comparison because BeeGFS supports switching between buffered I/O and direct I/O based on a fixed threshold value [5], whereas OrangeFS, in the tested implementation, only performs direct I/O on the client side [17].

This section uses abbreviations for various I/O and file system modes. `BIO` (buffered I/O) and `DIO` (direct I/O) refers to the I/O mode across several benchmarks, i.e., whether `O_DIRECT` was used with `open()`. File system modes refer to configuration changes applied to the three file systems used.

First, our new Lustre features can be toggled and configured independently, allowing us to investigate the impact of each. The following abbreviations for Lustre are used: 1. `vanilla` for the original Lustre version; 2. `autoIO` for the client-side decision algorithm; 3. `svrWB` for server-side write-back caching; 4. `delalloc` for the delayed (block) allocation; and 5. `XBatch` for cross-file batching of buffered writes.

For BeeGFS, we used two file cache modes on the clients [5]: 1. `buffered` mode representing the *default* file cache mode for write-back and read-ahead by using several static buffers; and 2. `native` mode that relies on the Linux page cache. BeeGFS's `native` mode offers the `tuneFileCacheBufSize` parameter (512 KiB by default) for switching to direct I/O above the threshold. In that case, all I/O operations bypass the page cache, communicating directly with the storage servers. `Native` is therefore comparable to our `autoIO`, which further considers additional parameters.

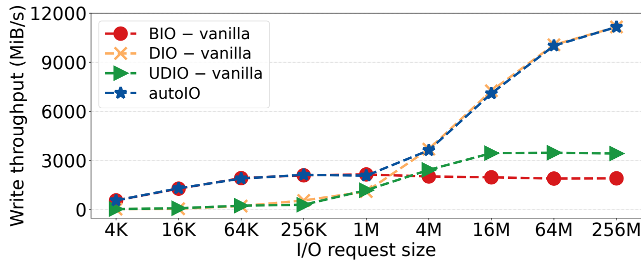
For OrangeFS, we used two server-side I/O modes: 1. `alt-aio` mode which is the *default* for accessing data on the storage servers by using buffered I/O via asynchronous I/O; and 2. `directio` mode that uses direct I/O to access data on the storage backend. Similar to our `svrWB` caching, which allows Lustre also to use buffered I/O on the servers, OrangeFS servers can therefore operate in buffered and direct I/O mode.

Our experiments were run on a Lustre cluster consisting of 4 MDTs, 8 OSTs, and 32 client nodes. The servers used a DDN AI400X2 Appliance backend (20 $\times$ SAMSUNG 3.84 TiB NVMe, 4 $\times$ IB-HDR100 100 Gbps), running Lustre version 2.15.58. All clients used an Intel Gold 5218 processor, 96 GiB of DDR4 memory, and ran CentOS 8.7 Linux. All nodes were interconnected using InfiniBand IB-HDR100.

BeeGFS used a storage architecture similar to Lustre, with the same hardware and configurations. Both clients and servers were running BeeGFS 7.4.0. For OrangeFS, we used version 2.10.0 running CentOS 8.7 Linux on both servers and clients. Each metadata storage target was configured with two metadata server instances, and each data storage target was configured with one data server instance. Thus, there are eight metadata instances and eight data server instances in total. The client kernel module for OrangeFS with CentOS 8.7 does not integrate with the Linux page cache, and all client I/Os (both direct and buffered I/O modes) are performed synchronously. Unless otherwise specified, a given file is striped across eight storage targets by default, and the stripe size is 1 MiB for all file systems.

4.1 Microbenchmarks

This section presents the experiments and results for various microbenchmark workloads using IOR and mdtest [1], which have become popular benchmarking tools in HPC [23].



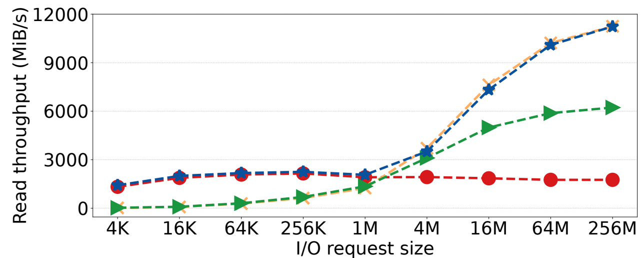


Figure 5: Single I/O stream throughput for Lustre.

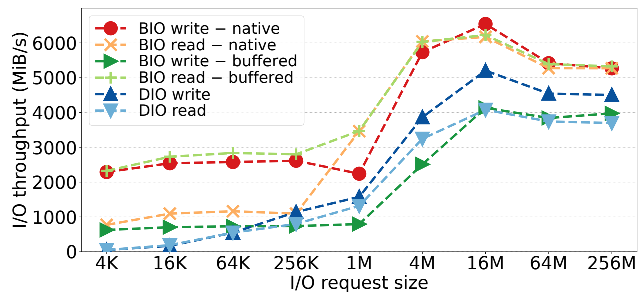


Figure 6: Single I/O stream throughput for BeeGFS.

Single I/O stream throughput First, we present the single I/O stream throughput for `BIO`, `DIO`, and unaligned `DIO` (`UDIO`) in vanilla Lustre and with our `autoIO` compared with BeeGFS and OrangeFS. In this case, IOR ran a single process, writing and reading data 2× the memory size with I/O sizes varying from 4 KiB to 256 MiB on the client.

Figure 5 visualizes the results that are overall similar to the local `ldiskfs`'s I/O throughput earlier in this paper (see Figure 1). The I/O sizes for unaligned direct I/O were increased by 8 bytes in this experiment as we aimed to observe whether aligning unaligned direct I/O can yield benefits over buffered I/O. As expected, the performance of unaligned direct I/O was lower than aligned direct I/O due to the overhead of extra memory allocation and copying. However, since it did not need to interact with page caching and management, it could outperform buffered I/O when the I/O size was larger than 4 MiB. Our autoIO mode took advantage of the two I/O modes and avoided their shortcomings, achieving the best overall performance over the entire range of I/O sizes.

Figure 6 shows the results for this workload running BeeGFS. BeeGFS achieved its highest throughput in `native` mode. Overall, BeeGFS achieved at most 6.5 GiB/s and 6.2 GiB/s for writes and reads, respectively. In comparison, Lustre's `autoIO` achieved 11.1 GiB/s (1.7×) and 11.2 GiB/s (1.8×) for writes and reads, respectively.

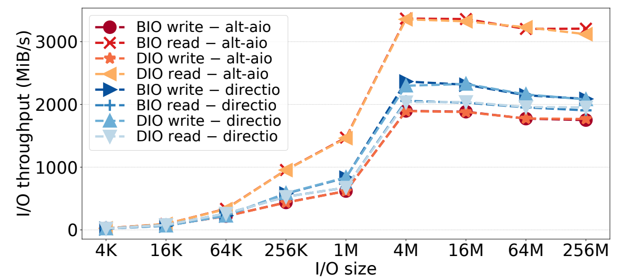


Figure 7: Single I/O stream throughput for OrangeFS.

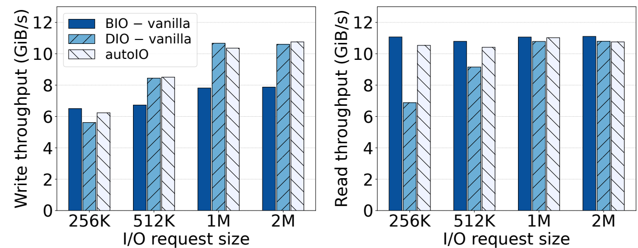


Figure 8: Lustre's I/O throughput for 16 processes.

OrangeFS reached the lowest I/O throughput of the three file systems (see Figure 7) with at most 2.3 GiB/s and 3.3 GiB/s for the respective writes and reads as all client I/Os were executed synchronously. Thus, the performance of direct I/O and buffered I/O are nearly the same. The results also indicate that the write performance using the server-side direct I/O mode (`directio`) has a slight edge over the `alt-aio` mode (buffered I/O). Read performance, on the other hand, achieved about a 50% higher throughput with `alt-aio` compared to direct I/O mode as it re-used data in the page cache.

Multiple I/O stream throughput Next, we ran IOR with 16 processes on a single client, sequentially writing and reading 80 GiB in file-per-process mode for I/O sizes ranging from 256 KiB to 2 MiB. Each file used only a single stripe object. The goal of this experiment was to investigate the trade-off phase of autoIO within the [256 KiB, 1 MiB] range and whether autoIO selects the best-performing I/O mode.

Figure 8 presents the results. Note that the I/O sizes in the range [256 KiB, 1 MiB] are larger than the small I/O threshold but smaller than the large I/O threshold. In this range, the efficiency of direct I/O and the performance optimizations due to write-back and read-ahead prefetching for buffered I/O are in a trade-off phase (see Algorithm 1), with `autoIO` almost reaching the best of both modes.

Table 2 lists the I/O statistics for the 512 KiB I/O size, that is, when autoIO switched to direct I/O due to memory

Table 2: I/O statistics for parallel I/O with 16 processes on 1 client node for a 512 KiB I/O size.

I/O Type	DIO (memory pressure)	DIO (cache overuse)	BIO (default)
Write	0	2,457,520	163,920
Read	45,593	5,993	4,738

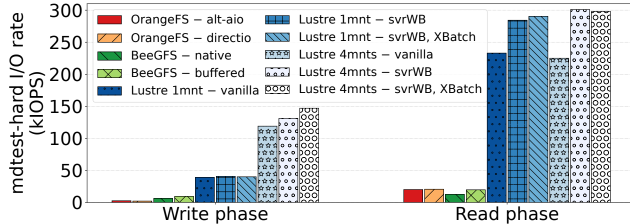


Figure 9: IO500's *mdtest-hard* performance for ten nodes.

pressure or cache over-usage. `Default` represents autoIO staying in buffered I/O mode. For instance, it shows that most writes were switched to direct I/O because more data was cached than allowed (see Section 3 in the page caches. For reads in autoIO, 45,593 I/Os were performed in direct I/O due to the memory pressure. In summary, these I/O statistics demonstrate that autoIO considers memory pressure and access locality to avoid excessive caching to obtain similar performances among the two I/O modes.

IO500's mdtest-hard workload To evaluate the I/O improvements for many small files, we ran mdtest in the *mdtest-hard* configuration of the IO500 "10-node challenge" benchmark. *mdtest-hard* generates many small files (with a size of 3091 bytes) in a single directory. We used 10 clients with 16 processes each, creating, writing, and reading 128,000 files per rank. Figure 9 shows the *mdtest-hard-write* and *mdtest-hard-read* results.

For Lustre, we compared several configurations: First, we used either one (`1 mnt`) or four separate Lustre mount points (`4 mnts`). For the latter, each mount point was assigned four MPI ranks. This technique mitigates locking congestion in the *virtual file system* (VFS) during parallel file creation in a shared directory. We also focused on measuring the impact of the `XBatch` and `svrWB` optimizations. Due to the many small files in this workload, autoIO and delayed allocation did not improve performance and are omitted. Finally, we used Lustre's *Data on MDT* (DoM) feature, which improves small file performance by placing small files only on the MDT and eliminating additional RPCs to the OSTs.

We also ran these experiments with BeeGFS (`native`, `buffered`) and OrangeFS (`alt-aio`, `directio`), albeit only working on a single mount point. In this configuration and compared to the vanilla Lustre case (`4 mnts`), they reached at most 2% of the performance for writes with OrangeFS

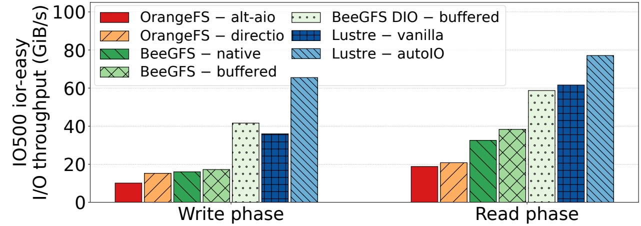


Figure 10: IO500's *IOR-easy* performance for ten nodes.

(`alt-aio`, `directio`) and 8% with BeeGFS (`buffered`). For reads, OrangeFS (`alt-aio`, `directio`) and BeeGFS (`buffered`) reached at most 9% of Lustre's performance.

For Lustre, we measured a performance benefit when our optimizations were enabled. In the write case (`4 mnts`) and with `svrWB` caching, the server sent a reply to the client as soon as the data was copied to the page cache, resulting in a 10% performance improvement over vanilla Lustre (`4 mnts`). With XBatch added, we achieved a 20% increase in performance. In the read case, the client's MPI ranks are offset so that the readers cannot benefit from the client's page cache. However, with `svrWB` caching enabled, the data is still available in the server's page cache, resulting in a 33% improvement over vanilla Lustre. For the `1 mnt` cases, the create-write performance did not benefit from the optimizations due to VFS locking congestion.

IO500's IOR-easy workload In this section, we evaluate the performance of all three file systems using IO500's *IOR-easy* use case with 10 nodes and 16 processes per node. Each process wrote and read in 16 MiB I/O size requests to a dedicated file for at least 300 seconds. Further, each file is striped across eight storage targets in all cases.

Figure 10 illustrates that `Lustre - autoIO` outperformed BeeGFS and OrangeFS in all configurations. This is because Lustre used direct I/O on both clients and servers with such a large I/O size (as opposed to `Lustre - vanilla` which used buffered I/O). BeeGFS used the less efficient buffered I/O on the servers for this I/O size, even when using direct I/O on the clients, resulting in inferior performance.

IO500's IOR-hard workload While *IOR-easy* represents a common sequential workload that generally works well for distributed file systems (and especially autoIO), *IOR-hard* creates a cyclic data distribution with an I/O size of 47,008 bytes that is neither aligned to page or file system block boundaries. Moreover, all processes access a single shared file with a segment count of 40,000. Figure 11 presents the results for the *IOR-hard* workload.

Generally, this is a challenging workload, particularly when aligning unaligned direct I/O and because of lock traffic overheads. In the case of Lustre and unaligned buffered writes, the client must first lock the object and read the unaligned page(s)

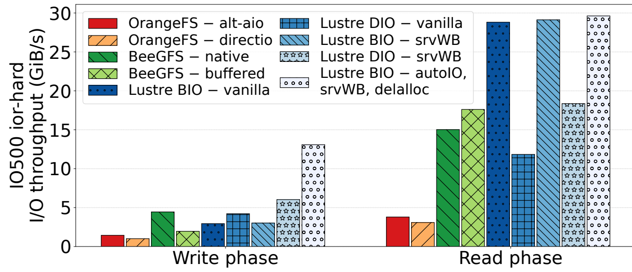


Figure 11: IO500's *IOR-hard* performance for ten nodes.

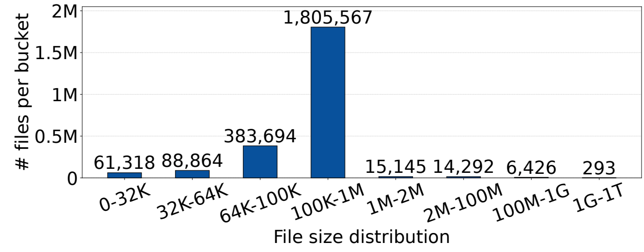


Figure 12: File size distribution of a large dataset.

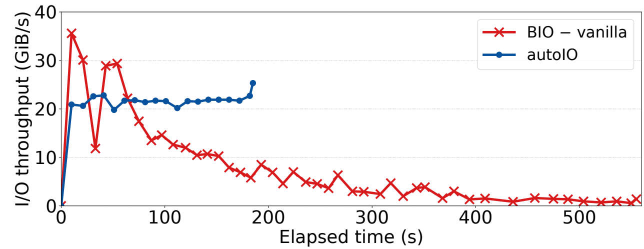


Figure 13: *dcp* I/O bandwidth over time for Lustre.

within the I/O range. Only then can the modified parts of the page(s) be updated. These unaligned RMW operations on the pages can severely affect performance. Although buffered writes can be aggregated in the page cache on a client, the writes must be contiguous on a single client and must be protected by the DLM extent lock. As the requested DLM extent lock must be page-aligned on the client, it may conflict with lock requests from other similarly page-aligned clients. This results in unnecessary lock contention when obtaining the DLM extent lock from the server.

Our I/O statistics revealed that even though buffered I/O accessed smaller file fragments, it still generated 3.5 million lock callbacks (35 per I/O segment). This is in contrast to unaligned direct I/O, which generated no lock callbacks due to server-side locking. As a result, the I/O throughput increased from 3 GiB/s to 4.2 GiB/s. By using `autoIO` as well as enabling `svrWB` (see Section 3.1) and `delalloc` (see Section 3.3), the write bandwidth reached 13 GiB/s.

Note that with server-side delayed allocation, data only needs to be written to the page cache without block allocation, thus reducing the I/O request latency. Moreover, we measured a reduction in file allocation fragments from 100K to about 35K, leading to a significant increase in 1 MiB size writes (100K+) that match our stripe size. Further, we enabled Lustre's overstriping feature [19, 38] which can improve I/O performance by allowing multiple stripes per OST. In this case, we set the `lfs setstripe` parameter to use 1,000 stripe objects, i.e., 125 stripes per OST. Overall, our additions improved performance by 4× compared with vanilla Lustre.

In the case of *ior-hard-read* and unaligned direct I/O reads, the I/O throughput increased from 11 GiB/s to 18 GiB/s with only `svrWB` caching enabled. This is due to many page cache read hits on the server that avoid reading from disk. The results also show that buffered I/O achieved much better read performance due to client-side read-ahead, reaching 30 GiB/s. With `autoIO`, reads achieved similar performances since the DLM lock for read operations from multiple clients were compatible. Therefore, the reads were performed in buffered I/O mode from the start.

Both BeeGFS and OrangeFS do not support unaligned direct I/O and used buffered I/O mode in this case. BeeGFS's native mode was slightly faster than Lustre with normal direct I/O. Lustre, on the other hand, with our `autoIO` and `delalloc`, was 2.9× faster than BeeGFS (`native`) for writes and 1.7× faster than BeeGFS (`buffered`) for reads.

4.2 mpiFileUtils/dcp workload

This section examines the I/O performance over time when copying a large file using *mpiFileUtils/dcp* [50]. Further, we copied an 8.8 TiB heterogeneous dataset, containing millions of files in a directory hierarchy with more than 10K directories (see the file size distribution in Figure 12), and compared the bandwidth with the three file systems in different modes. Dcp segments a large file into fixed-size chunks (4 MiB by default) and places a new distributed work item for each chunk in a global queue. The data copies are then distributed across multiple MPI ranks.

We first ran dcp on 32 nodes (16 processes each), writing and reading a 4 TiB to and from a single file (on the same file system) in parallel with a chunk size of 4 MiB. The source and target file were both striped across 8 OSTs. Figure 13 shows the bandwidth variation over time. Due to the large chunks that triggered direct I/O in the autoIO algorithm, it achieved a 3× performance improvement at 20 GiB/s and a more stable throughput than the buffered I/O performance in the vanilla case. As expected, the buffered I/O performance dropped drastically once most memory was consumed by the page cache. The observed lock callback count was over 77K. The memory pressure, due to interacting with the kernel's page cache, and false lock callbacks all contributed to the performance drop.

Next, we copied the entire dataset using dcp across 10 nodes (16 processes each) and a chunk size of 4 MiB. In this

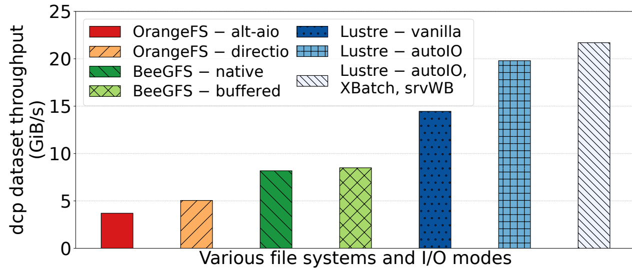
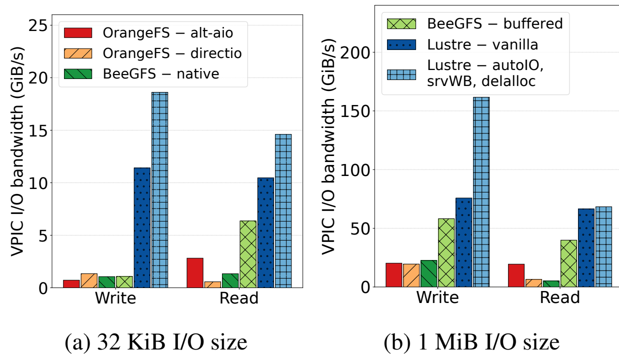


Figure 14: I/O throughput for copying a dataset with *dcp*.



(a) 32 KiB I/O size (b) 1 MiB I/O size

Figure 15: VPIC-IO bandwidth for various file systems.

case, we set autoIO's large I/O threshold to 512 KiB, which is similar to BeeGFS's default `native` mode.

Figure 14 presents the performance for the three file systems with various configurations. Overall, Lustre with `autoIO`, `XBatch`, and `svrWB` reached 21.7 GiB/s and outperformed vanilla Lustre by 1.5×, BeeGFS (`buffered`) by ∼2.6×, and OrangeFS (`directio`) by 4.3×. These results match the above *IOR-easy* conclusions in file system capabilities and demonstrate that our additions benefit both large files and heterogeneous datasets.

4.3 VPIC-IO workload

VPIC-IO is a macrobenchmark that represents the I/O kernel [12] of a large-scale plasma physics simulation that can compute, e.g., the reconnection and turbulence in solar weather. We ran the VPIC-IO kernel via h5bench [33] which uses an emulated compute time of two seconds for each time step and writes random particle data. Specifically, each MPI rank writes many particles into a single shared HDF5 file for a certain number of time steps, called *particle dump*.

We ran VPIC-IO on 32 nodes (16 processes each) for 8,192 and 262,144 particles and I/O sizes of 32 KiB and 1 MiB, respectively. In contiguous storage mode, the HDF5 metadata header is separate from the dataset data, with the data itself stored in one contiguous block in the HDF5 file. This led to MPI-IO starting not from 0 but at a specific offset (`mpi_off=2104` in our case), which is equal to the size of the metadata header. Since the offset of MPI-IO is not page-aligned, all I/O operations were unaligned. Figure 15 shows the write and read bandwidth for OrangeFS, BeeGFS, and Lustre. Lustre performed best among the three file systems. With `autoIO` and a 1 MiB I/O size, for instance, the write performance reached 3× of Lustre's `vanilla` performance as most I/O operations are switched to direct I/O due to lock contention.

Table 3: I/O statistics for VPIC-IO and 32 KiB I/O size.

Count	DIO (large I/O)	DIO (lock contention)	BIO (small I/O)	BIO (default)
AutoIO	0	807,876	1,043	11,324

The I/O statistics for the 32 KiB I/O size workload for Lustre and autoIO are listed in Table 3. Since the I/O size was small, I/O was initially handled in buffered I/O mode. Because I/O locks must be page-aligned in Lustre, this resulted in significant lock contention on the server, mainly due to unaligned I/O. This caused the clients to switch 807,876 I/O requests to direct I/O. 1,043 requests were handled in buffered I/O due to the small I/O size, and 11,324 were processed with buffered I/O by default. Finally, we monitored that more than 50% of the 32 KiB I/O requests were merged into a 1 MiB full stripe using `svrWB` caching and `delalloc`.

4.4 Nek5000 turbulent pipe flow workload

The Nek5000 application [20] for computational fluid dynamics (CFD) is a bulk-synchronous application. Its workflows can define step boundaries when Nek5000 should flush vector data, vector statistics, or write checkpoints. At each step, all ranks participate in writing to a single shared file at predefined offsets, depending on the number of participating processes.

In our experiments, we ran a turbulent pipe flow workload [47] on 32 nodes with 16 processes each. Contrary to VPIC-IO, this workload represents a real application and includes the computational component as well. In this experiment, Nek5000 executed 1,000 time steps (running for about 10 minutes), writing the corresponding vectors at each step, accounting for 600 GiB of data in total. Further, we used the Darshan profiling tool [40] to collect all I/O access sizes of

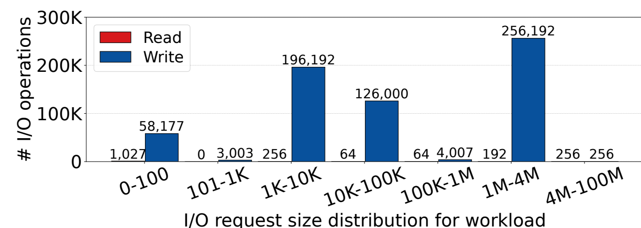


Figure 16: Nek5000's turbPipe workload I/O access sizes.

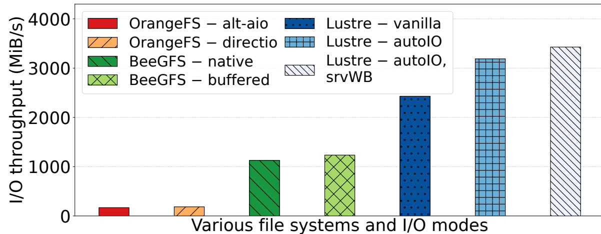


Figure 17: Nek5000 bandwidth for various file systems.

Table 4: I/O statistics for Nek5000's turbPipe workload.

Count	DIO (large I/O)	DIO (lock contention)	BIO (small I/O)	BIO (default)
AutoIO	128,000	132,000	372,281	65

this workload (see Figure 16), revealing a broad profile of small and large I/O requests.

Figure 17 presents the I/O bandwidth of Nek5000. Lustre with `autoIO` and `svrWB` caching reached 3,428 MiB/s and outperformed `vanilla` Lustre by 1.4×, BeeGFS (`buffered`) by ∼2.8×, and OrangeFS (`directio`) by 18.5×. Table 4 shows the I/O statistics using `autoIO`. The I/O throughput improved by more than 10% with `svrWB` caching enabled due to many small I/O requests. Note that the I/O sizes are proportional to the number of participating processes. Thus, with half the number of nodes and processes, e.g., 16 nodes, the I/O sizes become larger, increasing the effectiveness of direct I/O. In this case, Lustre with `autoIO` achieved 60% higher bandwidth than `vanilla` Lustre.

5 Related work

This section discusses the related work concerning avoiding the page cache, locking file system resources for strided I/O, dynamic I/O path, and I/O mode selection.

Direct access and page cache avoidance When direct I/O is used to bypass the kernel's page cache, it considerably impacts performance if used incorrectly while greatly benefiting large I/O operations [27]. Other approaches were made to directly access the storage device without involving the kernel to avoid overhead in the kernel's I/O stack [30,54]. Aerie [54], for example, processes metadata within a trusted metadata server in user space, potentially involving costly RPCs that can affect scalability.

For *non-volatile main memories* (NVMM) devices, page cache overhead due to data copies is unnecessary and can be directly addressed by the NVMM device. The DAX (direct access) feature in the Linux kernel uses the DAX interface to bypass the page cache and exposes the persistent memory driver [18,52]. Nevertheless, Simurgh [41] showed that DAX is often insufficient to expose the NVMM device's native performance. Other Linux features, e.g., `POSIX_FADV_DONTNEED fadvise()`, can reduce the impact of page caching for reads to discard a data range from the page cache after reads, avoiding page reclaiming during page allocation. A similar idea was developed for the Linux kernel with the `RMF_UNCACHED` flag, where pages are only added to the page cache for the duration of a read and removed after [16]. However, this approach still suffers from page management overhead, especially when using high-speed networks and storage backends.

Our work focuses on storage backends used by distributed file systems where bypassing the page cache is not always the best option. Therefore, we showed the importance of dynamically deciding whether to bypass the page cache based on both the I/O access patterns and the system state.

Cache strategies Numerous caching strategies targeting distributed file systems were designed over the years, including client-side write-back caching [45], I/O caching middlewares [60], employing machine learning to automate caching [24], or leveraging existing node-local storage devices [46]. Our implementation does not replace or add to these existing strategies. Instead, autoIO identifies requests for which it is more beneficial to bypass the client's page cache. AutoIO thus relieves cache pressure and allows cache strategies to be optimized for the remaining cached data. The break-even point at which direct I/O becomes useful depends not only on the given I/O size but also on lock contention, memory pressure, and overall access locality. AutoIO is the first transparent client-side algorithm to handle these parameters as part of the parallel file system.

Distributed locking Lock managers are vital to modern distributed software, e.g., to provide strong consistency in a distributed file system. In general, concurrent applications require locking that allows coordinated access to shared resources. Chubby [11] provides a coarse-grained synchronization mechanism between servers for reliability and availability in a loosely coupled distributed system. Lustre, on the other hand, must protect and coordinate access to shared resources by many clients in parallel to guarantee that both data and metadata remain consistent [9]. It further offers applications a user space API to request locks in advance for specific data ranges that are not allowed to expand [39]. This allows applications to avoid false lock conflicts beforehand and works well in strided I/O access patterns. Nonetheless, modifying applications to fully control their I/O behavior is not always feasible, especially when using I/O libraries. Our server-side adaptive locking can transparently react to lock-congested files without changing the application.

Dynamic I/O path Applications have widely different workload statistics that can even suddenly change with new

application types entering the HPC space, e.g., in the cases of data-driven application and AI workloads [10]. To accommodate past and future applications, parallel file systems often adopt the "one-size-fits-all" solution, such as Lustre [9], GPFS [49], and BeeGFS [25]. However, depending on the application and storage system, this can reduce I/O performance. Dynamic I/O paths can therefore be valuable to fit more closely to an application's I/O behavior.

Xiuqiao et al. [34] use a file handle-rich scheme in PVFS [14] providing a framework to enable a dynamic, fine-granular, and client-side I/O path section at runtime on a per-job basis. The *balanced placement I/O* (BPIO) library [56] intelligently allocates I/O paths for a parallel file system, binding a client to a storage target while evenly distributing the I/O traffic across components to proactively avoid contention points. To reduce I/O contention in an HPC environment, TAPP-IO [42] provides dynamic shared data placement mitigating resource contention and load-imbalance to improve application I/O, while *iez* [55] offers a transparent and adaptive control plane for balanced data placement.

Rather than focusing on data placement, we dynamically select the most suitable I/O path for each I/O request by using already existing I/O protocols, i.e., direct I/O, which can be challenging for developers to use directly.

Rigid vs. dynamic I/O mode Buffered I/O is still used as the default I/O mode in most situations. However, direct I/O is employed by other distributed file systems as well, e.g., BeeGFS and OrangeFS. BeeGFS's native mode, for instance, can switch from buffered I/O to direct I/O for I/O requests that are larger than the tunable `tuneFileCacheBufSize` parameter (512 KiB by default). Yet, direct I/O triggered on the client does not affect server behavior, which still relies on buffered I/O. Conversely, OrangeFS offers the *alt-aio* (default) and direct I/O modes that only affect server I/O behavior. The former uses a thread-based implementation for asynchronous I/O using `pread()` and `pwrite()`. Similarly, an NFS [22] server can use synchronous or asynchronous I/O (decided before launch). However, the cache protocols in BeeGFS, OrangeFS, and NFS are all non-coherent.

In contrast to the above file systems and to the best of our knowledge, we are the first to implement and evaluate a fully adaptive and transparent dynamic I/O path on both the file system client and server, using the most suitable I/O path in a given situation. In addition, we take file lock contention, memory pressure, and page cache reuse statistics into account to decide whether to use buffered I/O or direct I/O (even if unaligned) with strong consistency.

6 Conclusion and future work

This paper has presented a new approach to transparently and dynamically switch between buffered I/O and direct I/O in distributed file systems. We have shown the benefits of both I/O modes over a range of I/O sizes and have presented a client-side I/O mode switching algorithm that considers not only I/O sizes but also file lock contention and memory constraints. Other features include an adaptive server-side write-back cache, alignment of unaligned I/O, delayed allocation, and I/O request batching. We have achieved these features without compromising Lustre's strong consistency guarantees. Overall, our experimental results over several microbenchmarks, marcobenchmarks, and real-world workloads have shown that our approach reached up to 3× higher throughput than original Lustre and outperformed other distributed file systems by up to 13×.

Our future work covers several directions. First, we plan to conduct an extensive analysis of the performance impact of diverse I/O sizes, thresholds, file system configurations, and application workloads to further optimize and specialize autoIO's behavior. Second, we aim to modify autoIO's decision thresholds further so that they can automatically adapt depending on the system state by adopting previous work, e.g., machine learning-based prediction to optimize I/O bandwidth [6,53,58]. Third, we will design a server-side algorithm to switch between write-back and write-through modes that further considers the server state.

Acknowledgments and availability

We sincerely thank our shepherd Jinkyu Jeong and the anonymous reviewers for helping us improve our paper significantly.

This research was supported by the National Key R&D Program of China under Grant No.2022YFB4500304, the Natural Science Foundation of China under Grant No. 61832020 and No. 62332021.

This work was partially funded by the European Union's Horizon 2020 and the German Ministry of Education and Research (BMBF) under the "Adaptive multi-tier intelligent data manager for Exascale (ADMIRE)" project; Grant Agreement number: 956748-ADMIRE-H2020-JTI-EuroHPC-2019-1.

This work was also partially supported by the BMBF under the "Federated Digital Infrastructures for Research on Universe and Matter" FIDIUM project; Grant Agreement number: 05P21UMRC1.

References

[1] Ior and mdtest. https://github.com/hpc/ior, 2022. Accessed on Sep, 19, 2023.

[2] Orangefs. http://www.orangefs.org/, 2023. Accessed on Sep, 19, 2023.

[3] FAST '24: "Combining Buffered I/O and Direct I/O in Distributed File Systems" - Artifacts Description. https://doi.org/10.5281/zenodo.10425915, 2024.

[4] Abutalib Aghayev, Sage A. Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP), Huntsville, ON, Canada, October 27-30*, pages 353–369, 2019.

[5] BeeGFS. Client side caching modes. https://doc.beegfs.io/latest/advanced_topics/client_caching.html. Accessed on Sep, 19, 2023.

[6] Babak Behzad, Surendra Byna, Prabhat, and Marc Snir. Optimizing i/o performance of hpc applications with autotuning. *ACM Transactions on Parallel Computing (TOPC)*, 5(4):1–27, 2019.

[7] John Bent, Garth A. Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC), November 14-20, Portland, Oregon, USA*, 2009.

[8] Kevin J. Bowers, B. J. Albright, Lilan Yin, B. Bergen, and Thomas J. T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5):055703, 03 2008.

[9] Peter Braam. The lustre storage architecture. *CoRR*, abs/1903.01955, 2005.

[10] André Brinkmann, Kathryn M. Mohror, Weikuan Yu, Philip H. Carns, Toni Cortes, Scott Klasky, Alberto Miranda, Franz-Josef Pfreundt, Robert B. Ross, and Marc-Andre Vef. Ad hoc file systems for high-performance computing. *J. Comput. Sci. Technol.*, 35(1):4–26, 2020.

[11] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th Symposium on Operating Systems Design and Implementation (OSDI), November 6-8, Seattle, WA, USA*, pages 335–350, 2006.

[12] Suren Byna, Andrew Uselton, D Knaak Prabhat, and Yun He. Trillion particles, 120,000 cores, and 350 tbs: Lessons learned from a hero i/o run on hopper. In *Cray user group meeting*, 2013.

[13] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The next generation of ext2/3 filesystem. In *Linux Storage and Filesystem Workshop, February 12–13, 2007, San Jose, CA*, 2007.

[14] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *4th Annual Linux Showcase & Conference 2000, Atlanta, Georgia, USA, October 10-14*, 2000.

[15] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn M. Mohror, and Weikuan Yu. I/O characterization and performance evaluation of beegfs for deep learning. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP), Kyoto, Japan, August 05-08*, pages 80:1–80:10, 2019.

[16] Jonathan Corbet. Buffered i/o without page-cache thrashing. https://lwn.net/Articles/806980/, 2019. Accessed on Sep, 19, 2023.

[17] OrangeFS Documentation. Orangefs configuration file. https://docs.orangefs.com/configuration/admin_ofs_configuration_file/. Accessed on Sep, 19, 2023.

[18] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (ATC), Santa Clara, CA, USA, July 12-14*, pages 719–731, 2017.

[19] Patrick Farrell. Overstriping: Extracting maximum shared file performance. https://wiki.lustre.org/images/b/b3/LUG2019-Lustre_Overstriping_Shared_Write_Performance-Farrell.pdf, 2019. Accessed on Sep, 19, 2023.

[20] Paul Fischer, James Lottes, and Henry Tufo. Nek5000. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2007.

[21] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. Fasttrack: Foreground app-aware I/O management for improving user experience of android smartphones. In *2018 USENIX Annual Technical Conference (ATC), Boston, MA, USA, July 11-13*, pages 15–28, 2018.

[22] Thomas Haynes. Network file system (NFS) version 4 minor version 2 protocol. *RFC*, 7862, 2016.

[23] Michael Hennecke. Understanding daos storage performance scalability. In *Proceedings of the HPC Asia 2023 Workshops*, pages 1–14, 2023.

[24] Herodotos Herodotou. Autocache: Employing machine learning to automate caching in distributed file systems. In *35th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2019, Macao, China, April 8-12, 2019*, pages 133–139. IEEE, 2019.

[25] Frank Herold, Sven Breuner, and Jan Heichler. An introduction to beegfs. https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf, 2014. Accessed on Sep, 19, 2023.

[26] IBM. Considerations for the use of direct i/o (o_direct). https://www.ibm.com/docs/en/storage-scale/5.1.8?topic=applications-considerations-use-direct-io-o-direct, 2023. Accessed on Sep, 19, 2023.

[27] Russell Joyce and Neil C. Audsley. Exploring storage bottlenecks in linux-based embedded systems. *SIGBED Rev.*, 13(1):54–59, 2016.

[28] Linux Kernel. Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial. Accessed on Sep, 19, 2023.

[29] Dohyun Kim, Kwangwon Min, Joontaek Oh, and Youjip Won. Scalexfs: Getting scalability of XFS back on the ring. In *20th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 22-24*, pages 329–344, 2022.

[30] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. Nvmedirect: A user-space I/O framework for application-specific optimization on nvme ssds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), Denver, CO, USA, June 20-21*, 2016.

[31] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. Vaxclusters: A closely-coupled distributed system. *ACM Trans. Comput. Syst.*, 4(2):130–146, 1986.

[32] Julian M. Kunkel, John Bent, Jay Lofstead, and George S. Markomanolis. White paper: Establishing the io-500 benchmark. Technical report, The Virtual Institute for I/O, 2017.

[33] Tonglin Li, Suren Byna, Quincey Koziol, Houjun Tang, Jean Luca Bez, and Qiao Kang. h5bench: Hdf5 i/o kernel suite for exercising hpc i/o patterns. In *Proceedings of Cray User Group Meeting, CUG*, volume 2021, 2021.

[34] Xiuqiao Li, Limin Xiao, Meikang Qiu, Bin Dong, and Li Ruan. Enabling dynamic file I/O path selection at runtime for parallel file system. *The Journal of Supercomputing*, 68(2):996–1021, 2014.

[35] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Acclaim: Adaptive memory reclaim to improve user experience in android systems. In *2020 USENIX Annual Technical Conference (ATC), July 15-17*, pages 897–910, 2020.

[36] Linux man-pages 6.04. open(2) — linux manual page. https://man7.org/linux/man-pages/man2/open.2.html, 2023. Accessed on Sep, 19, 2023.

[37] Ajay Mohindra and Murthy V. Devarakonda. Distributed token management in calypso file system. In *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing (SPDP), Dallas, Texas, USA, October 26-29*, pages 290–297, 1994.

[38] Michael Moore. Exploring lustre overstriping for shared file performance on disk and flash. 2019.

[39] Michael Moore, Patrick Farrell, and Bob Cernohous. Lustre lockahead: Early experience and performance using optimized locking. *Concurrency and Computation: Practice and Experience*, 30(1), 2018.

[40] Nafiseh Moti, André Brinkmann, Marc-André Vef, Philippe Deniel, Jesús Carretero, Philip H. Carns, Jean-Thomas Acquaviva, and Reza Salkhordeh. The I/O trace initiative: Building a collaborative I/O archive to advance HPC. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, Denver, CO, USA, November 12-17, 2023*, pages 1216–1222. ACM, 2023.

[41] Nafiseh Moti, Frederic Schimmelpfennig, Reza Salkhordeh, David Klopp, Toni Cortes, Ulrich Rückert, and André Brinkmann. Simurgh: a fully decentralized and secure NVMM user space file system. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 46. ACM, 2021.

[42] Sarah Neuwirth, Feiyi Wang, Sarp Oral, and Ulrich Brüning. Automatic and transparent resource contention mitigation for improving large-scale parallel file system performance. In *23rd IEEE International Conference on Parallel and Distributed Systems (ICPADS), Shenzhen, China, December 15-17*, pages 604–613, 2017.

[43] Sarp Oral, Feiyi Wang, David Dillow, Galen M Shipman, Ross G Miller, and Oleg Drokin. Efficient object storage journaling in a distributed parallel file system. In *FAST*, volume 10, pages 1–12, 2010.

[44] Yoshihiro Oyama, Shun Ishiguro, Jun Murakami, Shin Sasaki, Ryo Matsumiya, and Osamu Tatebe. Reduction of operating system jitter caused by page reclaim. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS), Munich, Germany, June 10*, pages 9:1–9:8, 2014.

[45] Yingjin Qian, Wen Cheng, Lingfang Zeng, Marc-André Vef, Oleg Drokin, Andreas Dilger, Shuichi Ihara, Wusheng Zhang, Yang Wang, and André Brinkmann. Metawbc: Posix-compliant metadata write-back caching for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Dallas, TX, USA, November 13-18*, pages 56:1–56:20, 2022.

[46] Yingjin Qian, Xi Li, Shuichi Ihara, Andreas Dilger, Carlos Thomaz, Shilong Wang, Wen Cheng, Chunyan Li, Lingfang Zeng, Fang Wang, Dan Feng, Tim Süß, and André Brinkmann. LPCC: hierarchical persistent client caching for lustre. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Denver, Colorado, USA, November 17-19*, pages 88:1–88:14, 2019.

[47] Saleh Rezaeiravesh, Ricardo Vinuesa, and Philipp Schlatter. A statistics toolbox for turbulent pipe flow in nek5000. Technical report, KTH, Fluid Mechanics and Engineering Acoustics, 2019.

[48] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[49] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST), January 28-30, Monterey, California, USA*, pages 231–244, 2002.

[50] Danielle Sikich, Giuseppe Di Natale, Matthew LeGendre, and Adam Moody. mpifileutils: A parallel and distributed toolset for managing large datasets. Technical report, Lawrence Livermore National Lab (LLNL), Livermore, CA, USA, 2017.

[51] Chenlei Tang, Jiguang Wan, Yifeng Zhu, Zhiyuan Liu, Peng Xu, Fei Wu, and Changsheng Xie. Rafs: A raid-aware file system to reduce the parity update overhead for ssd raid. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1373–1378. IEEE, 2019.

[52] Nick-Andian Tehrany. *Evaluating Performance Characteristics of the PMDK Persistent Memory Software Stack*. PhD thesis, Vrije Universiteit Amsterdam, 2020.

[53] Abdul Jabbar Saeed Tipu. Hpc io and seismic data performance optimization using anns prediction based auto-tuning. 2023.

[54] Haris Volos, Sanketh Nalli, Sankaralingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: flexible file-system interfaces to storage-class memory. In *Ninth Eurosys Conference (EuroSys), Amsterdam, The Netherlands, April 13-16*, pages 14:1–14:14, 2014.

[55] Bharti Wadhwa, Arnab Kumar Paul, Sarah Neuwirth, Feiyi Wang, Sarp Oral, Ali Raza Butt, Jon Bernard, and Kirk W. Cameron. iez: Resource contention aware load balancing for large-scale parallel file systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rio de Janeiro, Brazil, May 20-24*, pages 610–620, 2019.

[56] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan S. Vazhkudai. Improving large-scale storage system performance via topology-aware and balanced data placement. In *20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), Hsinchu, Taiwan, December 16-19*, pages 656–663, 2014.

[57] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th Symposium on Operating Systems Design and Implementation (OSDI), November 6-8, Seattle, WA, USA*, pages 307–320, 2006.

[58] Li Xu, Thomas Lux, Tyler Chang, Bo Li, Yili Hong, Layne Watson, Ali Butt, Danfeng Yao, and Kirk Cameron. Prediction of high-performance computing input/output variability and its application to optimization for system configurations. *Quality Engineering*, 33(2):318–334, 2021.

[59] yongjoon. Rate limiter — sliding window counter. `https://medium.com/@avocadi/rate-limiter-sliding-window-counter-7ec08dbe21d6`, 2022. Accessed on Sep, 19, 2023.

[60] Dongfang Zhao, Kan Qiao, and Ioan Raicu. Hycache+: Towards scalable high-performance caching middleware for parallel file systems. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014*, pages 267–276. IEEE Computer Society, 2014.

A Artifact Appendix

Abstract

The *Artifacts Description* (AD) of this paper [3] provides detailed documentation of the used configurations for all file systems and experiments.

Scope

The AD has been made *available*. It includes detailed reference instructions to set up, deploy, and configure each used file system and each presented experiment. Please note that these artifacts are not functional due to the complexity of fully configuring and testing a Lustre installation automatically. However, the AD makes all instructions available for reference, making it possible to run similar experiments in similar configurations as provided in the paper.

Contents

The AD is sectioned into three main parts: Prerequisite information, e.g., software dependencies, installing the three used file systems, i.e., Lustre with autoIO, BeeGFS, and OrangeFS, and the description of all experimental workloads.

Prerequisites We describe the experimental setup and the required software dependencies for CentOS 8.4. Further, we provide requirements for setting up the cluster environment with the corresponding environment variables.

File system installation For Lustre, we provide detailed documentation of installing and configuring a Lustre parallel file system from scratch. We present information on Linux InfiniBand drivers (OFED) and provide reference Linux Bash scripts for installing and deploying the Lustre clients and servers (including autoIO). Moreover, we link the corresponding Git branches and pull requests to Whamcloud's issue and project tracking software (*JIRA*). These include all code changes from this paper and further serve as a reference for the status of each feature (eight in total).

Moreover, we provide the reference Bash scripts for installing the BeeGFS and OrangeFS clients and servers from scratch.

Experimental workloads We present the reference Bash scripts for each experiment and figure used in the paper. This includes setting file system configurations, e.g., enabling our server-side write-back, installing and running the evaluated applications. We further include a dedicated document for each experiment type in addition to the corresponding scripts.

Hosting

The AD is hosted on Zenodo and GitHub. GitHub supplements the long-term Zenodo repository and offers easy access to the artifacts' documentation and scripts. The corresponding versions are mirrored between Github and Zenodo, i.e., v1 on Zenodo mirrors v1.0 on GitHub. Please note that a DOI [3] via Zenodo is available for referencing these artifacts.