







A BeeGFS-Based Caching File System for Data-Intensive Parallel Computing

David Abramson^(✉) , Chao Jin , Justin Luong ,
and Jake Carroll 

The University of Queensland, St Lucia, QLD 4072, Australia
{david.abramson, c.jin, justin.luong,
jake.carroll}@uq.edu.au

Abstract. Modern high-performance computing (HPC) systems are increasingly using large amounts of fast storage, such as solid-state drives (SSD), to accelerate disk access times. This approach has been exemplified in the design of “burst buffers”, but more general caching systems have also been built. This paper proposes extending an existing parallel file system to provide such a file caching layer. The solution unifies data access for both the internal storage and external file systems using a uniform namespace. It improves storage performance by exploiting data locality across storage tiers, and increases data sharing between compute nodes and across applications. Leveraging data striping and meta-data partitioning, the system supports high speed parallel I/O for data intensive parallel computing. Data consistency across tiers is maintained automatically using a cache aware access algorithm. A prototype has been built using BeeGFS to demonstrate rapid access to an underlying IBM Spectrum Scale file system. Performance evaluation demonstrates a significant improvement in the efficiency over an external parallel file system.

Keywords: Caching file system · Large scale data analysis · Data movement

1 Introduction

In order to mitigate the growing performance gap between processors and disk-based storage, many modern HPC systems include an intermediate layer of fast storage, such as SSDs, into the traditional storage hierarchy. Normally, this fast storage layer is used to build a burst buffer that stages data access to the disk-based storage system at back-end [12, 16]. However, adding new tiers into the storage hierarchy also increases the complexity of moving data among the layers [17, 18].

The burst buffer can be provided on I/O or compute nodes of a cluster. The latter option, also called a node-local burst buffer [17, 18], equips each compute node with SSDs to decrease I/O contention to back-end storage servers. This leads to a deep hierarchical structure [12, 13] that contains, at the very least, a node private burst buffer and a shared external storage tier. To exploit hardware advances, many innovative software methods [5, 7, 16–18, 28–31] are proposed to utilize burst buffers efficiently. The management of node-local burst buffers has not been standardized. Some projects have investigated its use only for specific purposes, such as staging checkpoint data [7]

and caching MPI collective I/O operations [5]. Other projects, including BurstFS [28] and BeeOND [3], create a temporary file system on the private storage of compute nodes. However, these solutions manage the burst buffer independently of back-end storage, and programmers need to handle the complexity of moving data between storage tiers explicitly.

These tiers of persistent storage are typically used for different purposes in an HPC environment. Normally, the privately-owned internal storage maintains transient data to achieve faster I/O rates. In contrast, persistent data for long-term usage is stored externally, often using a parallel file system. Managing both tiers separately increases programming difficulties, such as maintaining data consistency and worrying the efficiency of moving data between the tiers. In order to bridge these layers, several challenges need to be addressed. First, the internal storage is isolated to individual compute nodes. Aggregating these siloed storage devices is necessary to provide scalable bandwidth for staging data more efficiently. Second, striping data across compute nodes is essential to accelerate parallel I/O for HPC applications. Third, programmers should be freed from having to move data explicitly between the storage tiers. Fourth, exploiting data access patterns through the storage layers can improve the performance of accessing the external parallel file system.

In this paper, we discuss the integration of the internal and external storage using a uniform solution. In particular, the paper describes a caching file system that automates data movement between a node-local burst buffer and a back-end parallel file system. It is realized by extending an existing parallel file system, BeeGFS [2]. Data access is unified across the storage layers with a POSIX-based namespace. In addition, the caching system improves storage performance by aggregating bandwidth of private storage, and exploiting data locality across the tiers. Furthermore, it increases SSD utilization by sharing data between compute nodes and across applications. Leveraging the inherent strengths of BeeGFS, such as data striping and meta-data partitioning, the caching extension supports high speed parallel I/O to assist data intensive parallel computing. Data consistency across storage tiers is maintained using a cache-aware algorithm.

Specifically, this paper presents the following contributions:

- A BeeGFS-based caching file system that integrates node-local burst buffers seamlessly with the back-end parallel file system;
- A unified data access abstraction that automates data movement and improves I/O performance by exploiting data locality across storage tiers;
- The caching extension mechanism that leverages parallel file system strengths to support scalable bandwidth and high-speed parallel I/O on the burst buffer.

The rest of this paper is organized as follows. Section 2 discusses related work and our motivation. Section 3 introduces the design and architecture of BeeGFS caching system. Section 4 presents the implementation details. Section 5 illustrates the performance evaluation of the prototype. Our conclusions follow in Sect. 6.

2 Background and Related Work

Most HPC systems adopt a hierarchical storage system [17, 18] to make the tradeoff between performance, capacity and cost. Recently, fast storage, such as SSDs, have been added between memory and disks to bridge the performance gap. This leads to a deep hierarchical structure. The top tier, such as the burst buffer [12, 16], provides high performance data access, and is placed close to compute nodes for containing actively used data. The bottom tier maintains long-term data persistently using disk-based solutions to provide high storage capacity. With most existing solutions, the software systems that manage different layers work separately [17, 18]. Accessing a disk-based storage tier has been standardized using a parallel file system, such as Lustre [26] and GPFS [22]. The appropriate way of managing a burst buffer is still under research [17, 18, 28, 29]. Currently, the internal storage layer cannot be directly utilized by most back-end parallel file systems [17, 18]. There is a lack of automatic data movement between storage tiers, and this causes a significant overhead to users [17, 18].

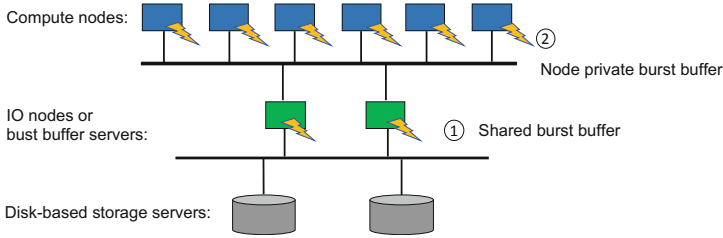


Fig. 1. Typical options of attaching a burst buffer.

2.1 Burst Buffer Overview

Currently, there are two major options to provide a burst buffer, as illustrated in Fig. 1. With the first option, compute nodes share a standalone layer of fast storage [16, 31, 33]. For example, the DoE Fast Forward Storage and IO Stack project [18] attaches the burst buffer to I/O nodes. Augmenting I/O nodes using SSDs improves bandwidth usage for disk-based external storage [31]. Cray DataWarp [10, 15] is a state-of-the-art system that manages a shared burst buffer, and it stages write traffic using a file-based storage space. Commands and APIs are supported for users to flush data from the burst buffer servers to the back-end file system. *Data elevator* [6] automates transferring data from the shared fast storage to the back-end servers. In addition, it offloads data movement from a limited number of burst buffer servers to compute nodes for scalable data transfer.

Efficiently organizing data for burst buffers has been investigated [29–31]. Data is typically stored in a log-structured format, while meta-data is managed for efficient indexing using Adelson-Velskii and Landis (AVL) tree, hash table, or a key-value store. Optimizing the performance of flushing data to the external storage is critical. I/O interference can be prevented by leveraging the scalability of distributed SSD array.

Controlling concurrent flushing orders [30] and orchestrating data transfer according to access patterns [14] have been proposed. SSDUP [23] improves SSD usage by only directing random write traffic to burst buffers.

With the second option, the burst buffer is privately owned by each compute node [17, 18]. This approach provides scalable private storage and further decreases I/O contention to the back-end storage [20]. Presently, the software that manages a node-local burst buffer is not standardized. There are mainly two ways of utilizing node-local burst buffers. One approach exploits fast local storage only for specific purposes. For example, locally attached SSDs are used to cache collective write data by extending MPI-IO [5], and to build a write-back cache for staging checkpoint data [7]. Another approach provides a general file system service. Research has shown that deploying a parallel file system on compute nodes can substantially reduce data movement to the external storage [34]. Distributed file systems, such as HDFS [24], have explored using host-local burst buffers to support aggregated capacity and scalable performance. These solutions are designed mainly for scalable data access, and lack of efficient support for high performance parallel I/O required by most HPC applications. The ephemeral burst-buffer file system (BurstFS) [28] instantiates a temporary file system by aggregating host-local SSDs for a single job. Similarly, BeeGFS On Demand (BeeOND) [3] creates a temporary BeeGFS [1] parallel file system on the internal storage assigned to a single job. These file system solutions enable sharing a namespace across compute nodes at front-end, but it is separated from the back-end file system. Therefore, users have to transfer data between the internal and external storage layers explicitly.

2.2 Uniform Storage Systems for HPC Storage Hierarchy

A few projects share the same goals with our work. UnivStor [27] provides a unified view of various storage layers by exposing the distributed and hierarchical storage spaces as a single mount point. UnivStor manages the address space using a distributed meta-data service and hides the complexity of moving data across storage layers. In addition, adaptive data striping is supported for moving data in a load balanced manner. Hermes [13] supports a caching structure to buffer data in the deep memory and storage hierarchy transparently. With Hermes, data can be moved seamlessly between different layers, from RAM and SSDs to disks. Hermes places data across storage layers according to access patterns and supports both POSIX and HDF5 [9] interfaces. In comparison, our approach takes advantage of an existing parallel file system to achieve a similar outcome. By extending BeeGFS, we provide a caching system to integrate a node-local burst buffer seamlessly with an external storage.

2.3 Parallel File System Overview

POSIX-based parallel file systems, such as Lustre [26], GPFS [22], and PVFS [4], are widely used to manage a disk-based back-end storage system. Typically, parallel data access and scalable bandwidth are provided by aggregating storage servers. Normally, data is striped across servers and meta-data is partitioned to accelerate parallel I/O. BeeGFS [1] is a parallel cluster file system with the POSIX interface. BeeGFS manages meta-data and files separately and its architecture consists of meta servers, storage

servers and management servers. BeeGFS transparently spreads data across multiple servers and scales up both system performance and storage capacity seamlessly. A single namespace is provided by aggregating all servers. File chunks are maintained by storage servers, whereas meta servers manage the meta-data, such as directories, access permission, file size and stripe pattern. Meta-data can be partitioned at the directory level such that each meta server holds a part of the file system tree. BeeGFS clients can communicate with both storage and meta servers via TCP/IP based connections or via RDMA-capable networks such as InfiniBand (IB). In addition, data availability is improved using built-in replication: buddy mirroring.

Managing a node-local burst buffer using a parallel file system can inherently leverage strengths, such as scalability and parallel data access, to assist data intensive computing. We extend BeeGFS to provide a caching system that bridges both internal and external storage tiers seamlessly. With the extension, BeeGFS allows moving data between the storage layers automatically. In addition, it improves data access performance by exploiting data locality across the storage tiers.

3 Design

The target environment consists of a compute cluster at the front-end and a persistent storage system at the back-end. Each compute node in the cluster is equipped with a large burst buffer, while the back-end storage system is managed using a POSIX-based parallel file system. Parallel applications running on compute nodes analyze data stored in an external file system. In order to decrease the I/O path of directly accessing the external system, hotspot data can be placed close to processors in the top tier of the storage hierarchy. Any applications running on the same cluster can access data stored in the burst buffer to reduce sharing data across programs using the external file system. Programmers are not required to know the exact location and long-term persistence for accessed files. In addition, to alleviate the performance gap between processors and storage, large files should be striped across compute nodes and serviced using parallel I/O. Moving data between the internal and external storage needs to be scalable with low I/O contention to avoid unnecessary network traffic.

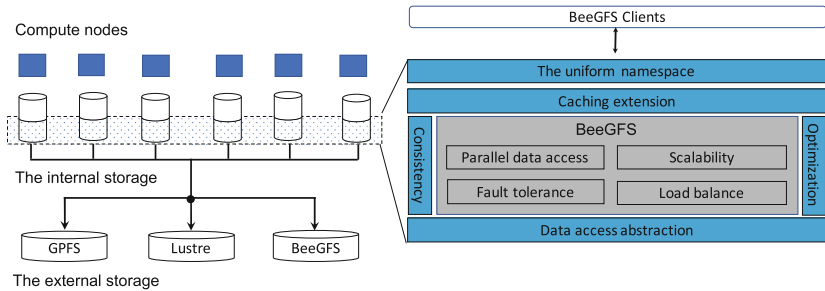


Fig. 2. The architecture of BeeGFS caching file system.

To meet these requirements, the fast storage isolated across compute nodes should be coordinated to provide a scalable caching pool. Each compute node contributes a part of its private storage and makes it accessible by other nodes. An instance of BeeGFS is deployed on the compute nodes to aggregate the siloed burst buffer. Managed by BeeGFS, the burst buffer stages data access for both write and read operations applied to back-end storage. Specifically, BeeGFS provides a parallel data service by accessing the targeted data set from an external file system. To improve performance, BeeGFS maintains most recently accessed files to avoid unnecessary network traffic and I/O to the back-end.

To provide a caching functionality, BeeGFS keeps track of accessed files. Whenever a file is requested, BeeGFS first verifies its existence and validity in the internal storage. In case a request cannot be satisfied due to a cache miss or an invalid copy, BeeGFS fetches data from the external file system transparently. Moving data, and examining its validity, are achieved using an on-demand strategy. If any updates need to be flushed to the external storage, BeeGFS synchronizes the permanent copy automatically.

Files are cached on compute nodes persistently, and are managed in a scalable manner by leveraging BeeGFS's scalability. In addition, BeeGFS organizes files with data striping and meta-data partitioning across the distributed fast storage to support high speed parallel I/O. When the free caching space is insufficient, least recently accessed files are evicted.

With the above design, programmers access files across storage tiers using a single namespace without worrying the exact data location, while data is committed for long-term usage automatically. Therefore, programmers are relieved from the complexity of manipulating data, but instead focusing on algorithm developments.

The architecture of BeeGFS caching system is illustrated in Fig. 2, which consists of two storage tiers. The top layer manages host-attached SSDs using BeeGFS. The bottom tier is the external storage cluster hosted by a parallel file system, such as GPFS, Lustre and others. To achieve the design targets, the following components extend BeeGFS to support the caching functionality:

- A POSIX-based uniform namespace: a uniform namespace across storage tiers enables accessing a piece of data regardless of its location. Most HPC applications rely on a traditional file interface. Therefore, providing a uniform namespace using the POSIX standard works with existing parallel applications seamlessly.
- Meta-data and data caching: files in the external file system are cached in the internal storage. BeeGFS maintains a consistent view of the back-end file system tree in the node-local burst buffer, and keeps track of cached objects by monitoring the existence and validity for each requested file and directory. It automatizes data movement across storage tiers, and exploits data locality to reduce unnecessary data traffic.
- Data access abstraction: moving data from the back-end file system can be achieved using file sharing. Each data site may be managed using different parallel file systems. The mechanism of accessing data should be applied to any file systems compliant with the POSIX standard. All of the data accessing details are hidden from users by the data access abstraction component.

- Data consistency: maintaining a coherent view between cached objects and their permanent copies needs to make an appropriate tradeoff between performance and consistency. Synchronizing updates should be optimized by avoiding unacceptable performance degradation.
- Optimization of data movement: moving data between the compute cluster and the external storage must be optimized with low I/O contention. Data transfer performance should be scalable with the number of involved compute nodes. In addition, data movement must take full advantage of high bandwidth and low latency of the storage network.

The performance target is to make both read and write operations applied to the external storage, with a cache hit, match the native BeeGFS on the burst buffer. With a cache miss, the read performance is restricted by the bandwidth of network and back-end storage. Accordingly, the extension should not change typical BeeGFS behaviors, such as high-performance data access, scalable storage capacity, load balancing and fault tolerance.

3.1 Uniform Namespace

The caching system provides a uniform namespace for accessing both internal and external files using the POSIX interface. Two sets of data are maintained in the internal storage: transient files and permanent files. The transient files require no long-term persistence, while each permanent file has a master copy in the external file system. Each file is referred using a local name, actually the full path. However, the local name for a permanent file also helps to identify its master copy in the external file system. This is achieved by linking an external directory to the internal file system, as illustrated in Fig. 3. In particular, each BeeGFS instance caches one external directory. The external director is specified when mounting the BeeGFS instance. The path name of the external directory is used to construct the external full path for each permanent file. Assume, a BeeGFS instance is mounted to the local directory */local/mounted* that caches files for an external directory */external/shared*. The local file */local/mounted/a.out* has an external copy */external/shared/a.out*, the name of which is produced by concatenating the external path, */external/shared*, and the relative path, *a.out*.

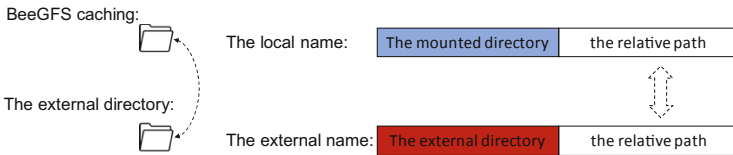


Fig. 3. Constructing the external path using the local name.

In another words, an internal directory is specified to hold the cache for an external data set. Actually, multiple external data sets, which may originate from different external file systems, can be linked to different internal directories. Therefore, the

POSIX file interface unifies storage access for both the internal burst buffer and external file systems. From the perspective of users, accessing the cached directory is no different than accessing other normal directories.

3.2 Caching Model

The caching model manages staging data for both read and write operations applied to the back-end parallel file system, and hides the details of moving data from users. In addition, it provides a consistent view on the shared external directory tree across storage tiers. For each cached object, its permanent copy maintained by the external file system is treated as the master version. To make an appropriate tradeoff between improving performance and enforcing data consistency, different strategies are applied to reading and writing files, and caching the namespace.

Writing files are staged using a write-back policy and reading files adopts a lazy synchronization method, in order to reduce unnecessary data movement. In contrast, the namespace is managed using an active strategy that guarantees a consistent global view across storage tiers. Reading the namespace is realized using an on-demand policy, while updating it is accomplished with a write-through method. The cache consistency is not controlled by the external file system, but actively maintained by the BeeGFS instance.

With the on-demand strategy, each level of the linked directory tree is cached only when it is traversed. When accessing a directory, all of its children directories are cached synchronously by duplicating its content to include name, creation time, update time, permission and size etc. However, files under the accessed directory are initially cached by only creating an empty position without copying the actual data. Subsequently, when the actual data is requested by any client, BeeGFS fetches the content to replace the empty position. Similar strategies are applied to synchronize updates made by the external file system.

To keep track of cached files and directories, BeeGFS meta-data, i.e. *inode*, is enhanced to include caching state and validity information. In addition, the creation and update times of the master copy are duplicated for consistency validation, the details of which is described in Sect. 3.3.

3.3 Data Consistency

The caching layer provides a two-level data consistency model to incorporate the performance difference between storage tiers. For concurrent data access applied to the internal storage layer, a strong and POSIX compliant consistency model is inherently supported by BeeGFS. Concurrent write operations can be coordinated by locking [2].

The caching model enforces data consistency between the cached objects and their permanent copies. Most scientific applications share data across clusters using a single writer model [1]. With this scenario, data is typically created by a single writer, even if it is shared with multiple writers across computer clusters. Accordingly, a weak consistency model is sufficient. The consistency is maintained per file. Validating the consistency is accomplished by comparing the update time between the cached object and its permanent copy. We assume each storage cluster uses a monotonically

increasing clock to identify time for an update operation. In addition, the compute cluster and the back-end storage cluster may hold different clocks at the same time. The update time of an external file is denoted as $mtime$. When creating a cached copy, $mtime$ is duplicated in its the meta-data, denoted as $mtime'$. During the lifetime of the cached object, $mtime'$ does not change. At the back-end, $mtime$ increases for each update applied to the permanent copy. Consequently, the validity of a cached object is examined using Eq. (1).

$$\begin{cases} \text{If } mtime' = mtime, \text{ the cached copy is valid.} \\ \text{If } mtime' < mtime, \text{ the cached copy is invalid.} \end{cases} \quad (1)$$

An invalid cached copy means that the master copy has been updated by the external file system. Therefore, synchronization is achieved by fetching the fresh copy from the external file system to replace the staled file in BeeGFS. This consistency semantic allows a single writer to spread its updates between multiple caching instances that share the same external directory tree.

3.4 Data Movement

Moving data across storage tiers should be parallelized to improve data transfer performance. Actually, data stored in the internal and external storage are both managed using parallel file systems. Files are striped across multiple servers and are serviced using parallel data access. Therefore, moving data across storage tiers should take advantage of both features. Instead of using any intermediate I/O delegates, each compute node should directly transfer file chunks that are managed by itself to the back-end storage. With this approach, the number of concurrent data transfer streams is scalable as the number of system nodes for both read and write operations. This type of highly parallel data movement can fully utilize the scalable bandwidth of storage network. In order to decrease I/O contention across files, transferring data can be ordered per file.

4 Implementation

The current prototype is implemented by augmenting the original meta-data and storage services. The meta server is extended to 1) keep track of each accessed object, 2) maintain data consistency between cached objects and their master copies in the external file system, and 3) coordinate staging data in and out of the internal storage. The storage server is improved to transfer data by leveraging BeeGFS data striping. The interaction of major caching components is illustrated in Fig. 4.

BeeGFS servers are implemented using C++, while its client is mainly written in C. BeeGFS clients, meta servers and storage servers communicate messages between each other using Unix sockets. Both meta and storage servers manage separate tasks using multiple worker threads. The caching extension expands the existing *inode* data structure and adds new messages and worker threads to achieve the design goal. The original BeeGFS structure is re-used as much as possible.

With the new BeeGFS caching system, both meta and storage servers are placed on compute nodes to manage the internal storage. Typically, one storage server is placed on each compute node, while the number of meta servers is configurable. The membership of BeeGFS cluster is maintained by a management service.

When mounting a BeeGFS instance, an external directory is linked, and it can be accessed using the Linux Virtual File System (VFS) interface. BeeGFS services VFS requests by accessing the external file system. For each VFS request, the BeeGFS client queries the meta-data service to determine if the target file exists internally. If an internal copy is valid, the request is serviced as normal. Otherwise, the meta server initiates moving the requested data to storage servers from the external file system.

4.1 Data Distribution

The caching extension re-uses the existing BeeGFS stripe formula to place all the chunks of a file across m storage servers in a round robin manner. Each cached file is uniformly partitioned into n chunks, and the size of each chunk is denoted $chunkSize$. The exact stripe formula is shown as Eq. (2):

$$offset(i) = i \times stripeSetSize + serverIndex \times chunkSize. \quad (2)$$

in which $stripeSetSize = m \times chunkSize$ and $offset(i)$ stands for the i^{th} stripe assigned to a storage server (identified by $serverIndex$).

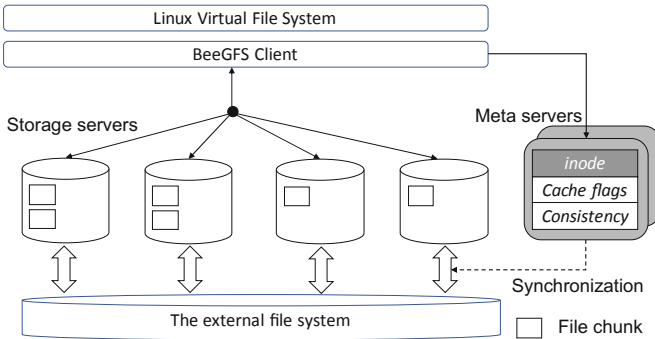


Fig. 4. The components of BeeGFS caching file system.

4.2 Meta Servers

The meta server coordinates storage servers to move data by adding new messages, such as *CachingFile*, and a worker thread *CacheEvictor*. The data structure that keeps track of cached objects must be persistent, otherwise, the caching system may become inconsistent in case of failures. Therefore, the existing BeeGFS data structures are re-used by leveraging its serialization logic to preserve included caching information persistently. The BeeGFS *inode* structure contains essential information, such as an

entry id, which is used to identify each file and directory, the placement map for file chunks, and a feature field used for buddy mirroring. The *inode* structure is augmented to include the caching state for each file and directory, and to identify if the cached object is up-to-date. The feature field is extended to represent two flags: *caching* and *dirty*. The caching flag indicates if the associated file has a copy in the internal storage. *Caching is off* means that the file is created just for holding a position or has been evicted. After all the chunks of a file are duplicated in BeeGFS, *caching* is set *on*. The *dirty* flag is set when any update is applied to the cached copy. The master copy's *mtime* is also duplicated into *inode* for verifying the validity of a cached copy.

Namespace Consistency. Namespace coherence is maintained transparently using a polling approach to detect changes made by the external file system. However, an aggressive polling approach that periodically verifies the entire cached namespace causes a significant overhead for a deep directory tree. To implement an on-demand policy of enforcing consistency, a lazy polling approach is adopted that only examines the part of file system tree being traversed.

In particular, *stat*, *open*, *lookup* and *readdir* operations are intercepted. The external path name is reconstructed to validate the existence of the target item. If any new directory is detected, its content is cached immediately. For any new file created in the external directory, an internal entry is instantiated without copying the actual data. In addition, its *caching* flag is set *off* to indicate subsequent synchronization is required.

As described previously, updates applied to the internal directory tree are synchronized with the external file system using a write-through policy. Updates generated by operations, such as *chmod*, *chgrp*, *mv*, *rm* etc., are replicated to the back-end file system simultaneously. For a new file or directory created in BeeGFS caching, the external file system immediately holds a position for them. But the actual content is synchronized when required. BeeGFS exclusively partitions meta-data across multiple meta servers. Updates from different meta servers cause no conflicts.

Verifying namespace consistency changes the default behavior of read only meta-data operations, such as *stat*, *lookup* and *readdir*. These operations make no changes to the namespace in the original BeeGFS. However, with the caching extension, these operations may detect modifications on the external namespace, the synchronization of which causes updating the namespace cached in the internal storage.

File Consistency. File consistency is maintained by intercepting the *open* operation. Upon opening a file, the meta server queries its caching flag for the internal copy. In case the cached copy is present, its validity is examined using Eq. (1). If necessary, the master version is copied to replace the local stale one, which is coordinated by the meta server using a caching request. To avoid conflicts, multiple simultaneous *open* operations applied to the same file are serialized by the meta server. With this serialization, only a single caching request is created for one *open* operation and all other operations applied to the same file block until the requested file is ready to access. Synchronizing a file needs to update chunks that are distributed across storage servers. During the process, the target file should not be accessed, because its content may belong to different versions. Therefore, locking is used to protect file synchronization.

The transaction of moving or updating a file typically involves multiple storage servers. The exact process consists of two stages: 1) notifying all of involved storage

servers and 2) moving file chunks. In the first stage, a *CachingFile* message is sent to all of the involved storage servers. The exact message includes file name, file size, and data stripe pattern etc. This stage is protected using a read lock. After sending the request, the second stage waits to start until all of the storage servers respond. At the end of the first stage, the read lock is released and a write lock is obtained immediately for the second stage. Both locks prevent other threads from opening the same file for updates until all the chunks have been successfully synchronized. After the secondary stage completes, the *open* operation continues as normal.

Optimization. Identifying an external file requires the concatenation of its internal path with the name of cached external directory, as illustrated in Fig. 3. However, reconstructing a path name in BeeGFS is not straightforward. BeeGFS does not keep a full path for any file or directory. In addition, meta-data for each entry is stored in a separate object, and each file is identified using its *entry id* and parent directory. Therefore, constructing the path name for a file or directory must look up each entry's parent backwards by going through a number of separated objects, which is time-consuming as it may require reloading the entry from storage. To improve the efficiency of verifying data consistency, constructing a path name is accelerated. When looking up a file from the root level, each parent entry is kept in memory for subsequent path construction.

4.3 Storage Servers

To assist file caching, eviction, and synchronization operations, BeeGFS storage servers are coordinated by the meta server. With file chunk distribution, each storage server only keeps a part of a cached file, and the storage server maintains each chunk using a local file. Upon receiving the request of transferring a file, the storage server creates the working directory on the internal storage and then initiates copying file chunks. Each storage server transfers data by only accessing a region of the target file from the external file system, instead of going through the whole file. In order to improve performance for accessing a file partially, instead of using *lseek*, *read* and *write* system calls, *pread* and *pwrite* are adopted. In addition, storage I/O access to the external file system must be efficient. The remote file is accessed using the recommended block size, which is detected using *stat*. Therefore, the exact data transfer is realized using a block-based algorithm, as shown in Algorithm 1.

Buddy Mirror. BeeGFS supports buddy mirroring to improve data reliability. Each group of buddy mirrors consists of two servers: the primary and secondary, in which each secondary server duplicates its primary counterpart. When the primary and secondary copies become inconsistent, it is required to synchronize buddies, which is called *resync*.

The caching module takes advantage of buddy mirroring to improve data availability, which is configurable, and to increase bandwidth for hotspot files. Presently, the replication for data caching is performed asynchronously such that the primary server does not wait until the secondary one finishes the caching request. However, the

caching request must avoid interfering a resync process of buddy mirror. Specifically, caching requests are serviced until a resync process is completed.

Algorithm 1. The block-based data transform algorithm on the storage server.

```

1  procedure BLOCKIO (fileDesc, buffer, len, offset, blocksize, isRead)
2      total = 0
3      bytes = 0
4      while total ≤ len do
5          if len − total < blocksize then
6              iosize = count − total
7          else
8              iosize = blocksize
9          if isRead
10             bytes = pread (fileDesc, buffer + total, iosize, offset + total)
11         else
12             bytes = pwrite (fileDesc, buffer + total, iosize, offset + total)
13         if bytes ≤ 0 then return error
14         total = total + bytes
15     return success

```

4.4 Cache Eviction

When the free caching space is insufficient, some less accessed files should be evicted. Clean copies that are not updated in the caching, can be deleted directly. In contrast, for other dirty copies, updates should be flushed to the external file system.

The cache eviction routine is implemented by adding a worker thread, *CacheEvictor*, to the meta-data service, which is launched on startup with other worker threads. This eviction thread periodically selects less accessed files from storage servers that are low in space and moves them out of BeeGFS to keep available free space as required. The storage usage report created by the management service is re-used to detect the whole system storage usage. The management service monitors storage usage for each server and classifies them into *emergency*, *low* and *normal* capacity groups. The storage usage report is collected for each storage server periodically and sent to the meta servers. With this report, a Least Recently Used (LRU) policy is adopted to make decisions on which files should be moved out. Upon eviction, flushing dirty copies uses the same block-based data transfer algorithm as described in Sect. 4.3. A write lock is acquired to guarantee the eviction process is not interrupted by normal file operations.

5 Performance Evaluation

The prototype was built on BeeGFS version 6.1 and it was evaluated on the FlashLite system at the University of Queensland [8]. FlashLite contains large amounts of main memory and high-speed secondary storage, SSDs. The back-end storage is provided by an IBM Spectrum Scale (GPFS) system, and all compute nodes communicate with the GPFS system using the native Network Shared Disk (NSD) protocol [25]. High performance networking, such as Dual rail 56Gbps Mellanox InfiniBand fabric, connects FlashLite and GPFS servers. Each compute node of FlashLite has the following system configuration:

- 2 × Xeon E5-2680v3 2.5 GHz 12core Haswell processors;
- 512 GB DDR4-2133 ECC LRDIMM memory (256 GB per socket);
- 3 × 1.6 TB Intel P3600 2.5" NVMe (SSD) drives of internal storage;
- 1 TB RAID 1 system disk;
- 2 × Mellanox 56 Gb/s FDR Single Port InfiniBand adapter.

The CentOS 7 operating system, with kernel version 3.10.0–693, is installed on each node that manages SSDs using a standard *ext4* filesystem. The BeeGFS caching system was deployed for performance evaluation on 6 compute nodes of FlashLite. The system was installed with one meta server, one management server, and 6 storage servers. One BeeGFS storage server was placed on each compute node, while one compute node was selected to run both meta and management servers. The BeeGFS file system was mounted on each node at */mnt/beegfs* for caching a remote directory in GPFS. RDMA is enabled across the servers using the default BeeGFS OpenTk communication library. File chunk size was set to 512 KB, and a striping pattern RAID0 using four targets of storage server was specified. Buddy mirroring was disabled during the experiment. Performance was evaluated for both meta-data operations and file data accesses.

5.1 Meta-Data Performance

The performance of meta-data operations was evaluated using MDTtest [19]. MDTtest measures meta-data performance through a series of *create*, *stat* and *delete* operations on a tree of directories and files. The operations were conducted in parallel on up to 6 compute nodes, in which each node run one MDTtest instance. We compared these operations for three different situations: GPFS, BeeGFS caching prototype, and the original BeeGFS system (version 6.1). The vanilla BeeGFS system was installed on the same set of compute nodes in which the caching prototype was deployed, and was instantiated with the same configuration. MDTtest was configured with a branch factor of 3, and a depth of 3. The number of items per tree node was set to 100, for a total of 4,000 files/directories per task. Each situation was evaluated using the number of performed transactions per second as metrics. The averaged value with a standard deviation was collected.

For read-only meta-data operations, such as *stat* for files and directories illustrated in Fig. 5, vanilla BeeGFS performs faster than GPFS, because it is deployed on internal storage. However, for write-intensive operations, such as creation and deletion of files

and directories, as shown in Fig. 6 and Fig. 7 respectively, GPFS performs better than vanilla BeeGFS. This is because BeeGFS was created with only one meta-data server, which is not scalable for highly concurrent meta-data operations.

Overall, the caching prototype performs the worst for both read- and write-intensive meta-data operations. This is because the caching system not only conducts operations on internal storage, but also replicates these operations on the back-end storage. Our prototype performs both operations in a sequential manner, and this degrades performance. However, as shown in Sect. 5.2, the performance degradation has a negligible impact on the speed of accessing data in internal SSDs because meta-data operations only compose a tiny fraction of data access activities. Future work will investigate how to improve meta-data operations by maintaining consistency asynchronously.

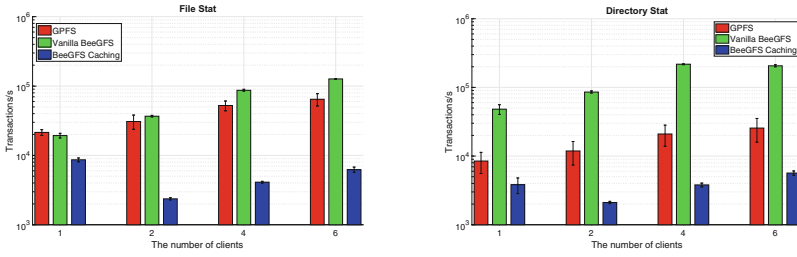


Fig. 5. MDTest file and directory stat.

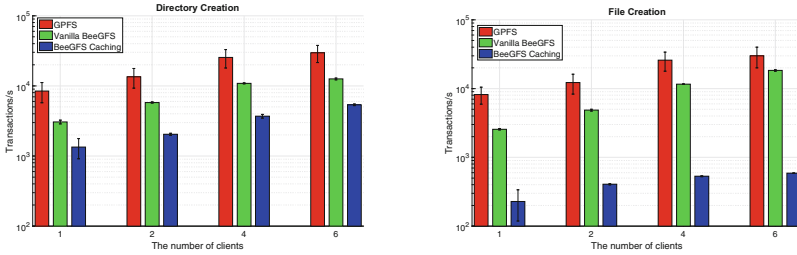


Fig. 6. MDTest file and directory creation.

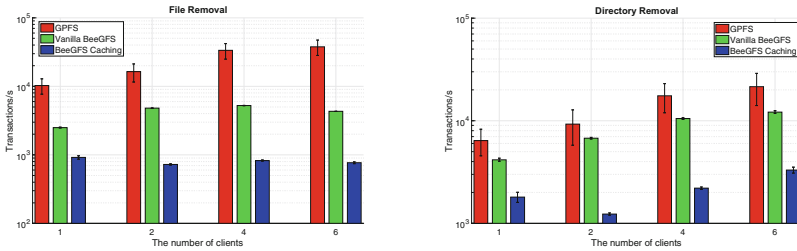


Fig. 7. MDTest file and directory removal.

5.2 Data Performance

Interleaved or Random (IOR) [10] was performed on the same set of compute nodes to evaluate the performance of accessing files stored in GPFS via the caching prototype. We compared two scenarios: cache miss and cache hit, for different file sizes, from 100 MB to 100 GB. One IOR client was placed on each compute node, while up to 6 IOR clients were used during the experiment. When a cache miss occurs, the requested file is fetched from back-end GPFS, while a cache hit means the requested file already stays in the BeeGFS caching system. In order to amortize the disturbance of other workloads present on GPFS, the IOR experiments were repeated over 24 h at hourly intervals. For testing read operations, the tested files were generated in advance and flushed out of the internal storage to enforce the behavior of cache-miss. The aggregated bandwidth perceived by multiple IOR clients was collected. The averaged values with a standard deviation were shown.

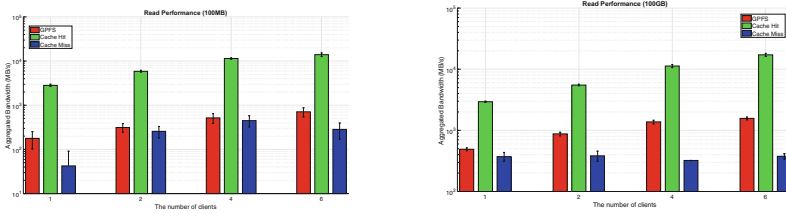


Fig. 8. IOR read performance.

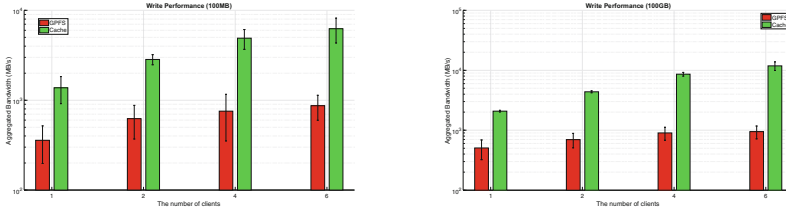


Fig. 9. IOR write performance.

Overall, the experiment shows that accessing data from the caching layer is significantly faster than directly accessing GPFS for both read and write operations, regardless of data size. In addition, accessing 100 GB large files delivers higher bandwidth than 100 MB files due to more efficient sequential operations on both internal and external storage. The performance of reading data from GPFS and the caching prototype is shown in Fig. 8, while Fig. 9 illustrates writing performance. The caching prototype provides scalable data access with the number of clients for both read and write operations. However, with a cache miss, the BeeGFS caching system is

slower than GPFS because the requested data need to be copied into the internal storage first before being forwarded to applications. Therefore, the cache miss introduces an extra overhead in comparison to accessing GPFS directly. Future work will explore how to overlap data transfer across storage tiers to hide the extra latency for cache miss cases.

6 Conclusions

In order to improve storage performance, many HPC systems include an intermediate layer of fast storage, such as SSDs, between memory and the disk-based storage system. In particular, compute nodes may contain a large amount of fast storage for staging data access to the back-end storage. Frequently, this layer of node-local burst buffer is managed independently of the back-end parallel file system. To integrate the node-local burst buffer seamlessly with the existing storage hierarchy, we extend BeeGFS to provide a caching file system that bridges both internal and external storage transparently. Data access to the burst buffer and the back-end parallel file system is unified using a POSIX-based namespace. Moving data between the internal and external storage is automated and long-term data persistency is committed transparently. Accordingly, users are released from the complexity of manipulating the same piece of data across different storage tiers. In addition, the extension investigates how to utilize the burst buffer by leveraging the strengths of a parallel file system to accelerate data-intensive parallel computing. Taking advantage of BeeGFS, scalable I/O bandwidth is provided by aggregating siloed fast storage, and storage performance is improved by exploiting data locality across storage tiers. Data striping across storage servers not only supports high performance parallel IO, but also scales data transfer between storage tiers. In addition, a block-based algorithm increases the efficiency of data movement. The performance evaluation demonstrates that BeeGFS caching system improves data access significantly over directly accessing GPFS for both temporal and spatial locality patterns. However, the present prototype imposes additional overhead on meta-data operations due to maintaining data consistency between storage tiers synchronously. Our future work will explore how to reduce the extra overhead and apply the extension mechanism to other general parallel file systems.

Acknowledgements. We thank HUAWEI for funding this research project. We also acknowledge the University of Queensland who provided access to FlashLite. FlashLite was funded by the Australian Research Council Linkage Infrastructure Equipment Fund (LIEF).

References

1. Abramson, D., Carroll, J., Jin, C., Mallon, M.: A metropolitan area infrastructure for data intensive science. In: Proceedings of IEEE 13th International Conference on e-Science (e-Science), Auckland (2017)
2. BeeGFS. <https://www.beeufs.io/content/>. Accessed 21 Nov 2019
3. BeeOND. <https://www.beeufs.io/wiki/BeeOND>. Accessed 21 Nov 2019
4. Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: PVFS: a parallel file system for Linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, pp. 317–327. USENIX, Atlanta (2000)
5. Congiu, G., Narasimhamurthy, S., Süß, T., Brinkmann, A.: Improving collective I/O performance using non-volatile memory devices. In: Proceedings of 2016 IEEE International Conference on Cluster Computing (CLUSTER 2016). IEEE, Taipei (2016)
6. Dong, B., Byna, S., Wu, K., Prabhat, J.H., Johnson, J.N., Keen, N.: Data elevator: low-contention data movement in hierarchical storage system. In: Proceedings of 23rd IEEE International Conference on High Performance Computing (HiPC). IEEE, Hyderabad (2016)
7. Dong, X., Xie, Y., Muralimanohar, N., Jouppi, N.P.: Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Trans. Architect. Code Optim. (TACO)* **8**(2), 6:1–6:29 (2011)
8. FlashLite. <https://rcc.uq.edu.au/flashlite>. Accessed 21 Nov 2019
9. Folk, M., Heber, G., Koziol, Q., Pourmal, E., Robinson, D.: An overview of the HDF5 technology suite and its application. In: Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases, pp. 36–47. ACM, Uppsala (2011)
10. IOR. <http://wiki.lustre.org/IOR>. Accessed 21 Nov 2019
11. Henseler, D., Landsteiner, B., Petesch, D., Wright, C., Wright, N.J.: Architecture and design of cray datawarp. In: Proceedings of 2016 Cray Users’ Group Technical Conference (CUG 2016). Cray, London (2016)
12. He, J., Jagatheesan, A., Gupta, S., Bennett, J., Snavey, A.: DASH: a recipe for a flash-based data intensive supercomputer. In: Proceedings of the 23rd ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2010), pp. 1–11. IEEE, New Orleans (2010)
13. Kougkas, A., Devarajan, H., Sun X.-H.: Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system. In: Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2018), pp. 219–230. ACM, Tempe (2018)
14. Kougkas, A., Dorier, M., Latham, R., Ross, R., Sun, X.-H.: Leveraging burst buffer coordination to prevent I/O interference. In: Proceedings of 12th IEEE International Conference on e-Science (e-Science 2017). IEEE, Baltimore (2017)
15. Landsteiner, B., Pau, D.: DataWarp transparent cache: implementation, challenges, and early experience. In: Proceedings of 2018 Cray Users’ Group Technical Conference (CUG 2019). Cray, Stockholm (2018)
16. Liu, N., et al.: On the role of burst buffers in leadership-class storage systems. In: Proceedings of 28th IEEE Symposium on Mass Storage Systems and Technologies (MSST 2012). IEEE, San Diego (2012)
17. Lockwood, G.K., Hazen, D., Koziol, Q., et al.: Storage 2020: a vision for the future of HPC storage. Lawrence Berkeley National Laboratory (LBNL) Technical report, No. LBNL-2001072. NERSC (2017)

18. Lofstead, J., Jimenez, I., Maltzahn, C., Koziol, Q., Bent, J., Barton, E.: DAOS and friends: a proposal for an exascale storage system. In: Proceedings of the 29th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2016), pp. 807–818. IEEE, Salt Lake City (2016)
19. MDTest. <http://wiki.lustre.org/MDTest>. Accessed 21 Nov 2019
20. Nisar, A., Liao, W.-K., Choudhary, A.: Scaling parallel I/O performance through I/O delegate and caching system. In: Proceedings of the 21st ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2008). IEEE, Austin (2008)
21. Ovsyannikov, A., Romanus, M., Straalen, B., Weber, G.H., Trebotich, D.: Scientific workflows at DataWarp-speed: accelerated data-intensive science using NERSC’s burst buffer. In: Proceedings of 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS 2016), pp. 1–6. IEEE, Salt Lake City (2016)
22. Schmuck, F., Haskin, R.: GPFS: a shared-disk file system for large computing clusters. In: Proceedings of 1st USENIX Conference on File and Storage Technologies (FAST 2002). USENIX, Monterey (2002)
23. Shi, X., Li, M., Liu, W., Jin, H., Yu, C., Chen, Y.: SSDUP: a traffic-aware SSD burst buffer for HPC systems. In: Proceedings of the International Conference on Supercomputing (ICS 2017). ACM, Chicago (2017)
24. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST 2010). IEEE, Incline Village (2010)
25. Volobuev, Y.: GPFS NSD Server Design and Tuning, IBM GPFS Development Document, Version 1.0 (2015)
26. Wang, F., Oral, S., Shipman, G., Drokin, O., Wang, T., Huang, I.: 2010 Understanding Lustre Filesystem Internals, Oak Ridge National Laboratory Technical report, No. ORNL/TM-2009/117. National Center for Computational Sciences (2009)
27. Wang, T., Byna, S., Dong, B., Tang, H.: UniVistor: integrated hierarchical and distributed storage for HPC. In: Proceedings of 2018 IEEE International Conference on Cluster Computing (CLUSTER 2018). IEEE, Belfast (2018)
28. Wang, T., Mohror, K., Moody, A., Sato, K., Yu, W.: An ephemeral burst-buffer file system for scientific applications. In: Proceedings of the 29th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2016), pp. 807–818. IEEE, Salt Lake City (2016)
29. Wang, T., et al.: MetaKV: a key-value store for metadata management of distributed burst buffers. In: Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2017), pp. 807–818. IEEE, Orlando (2017)
30. Wang, T., Oral, S., Pritchard, M., Wang, B., Yu, W.: TRIO: burst buffer based I/O orchestration. In: Proceedings of 2015 IEEE International Conference on Cluster Computing (CLUSTER 2015). IEEE, Chicago (2015)
31. Wang, T., Oral, S., Wang, Y., Settlemyer, B., Atchley, S., Yu, W.: BurstMem: a high-performance burst buffer system for scientific applications. In: Proceedings of the 2014 IEEE International Conference on Big Data (Big Data 2014). IEEE, Washington (2014)
32. Xie, B., et al.: Characterizing output bottlenecks in a supercomputer. In: Proceedings of the 25th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2012). IEEE, Salt Lake City (2012)

33. Zhang, W., et al.: Exploring memory hierarchy to improve scientific data read performance. In: Proceedings of 2015 IEEE International Conference on Cluster Computing (CLUSTER 2015), pp. 66–69. IEEE, Chicago (2015)
34. Zhao, D., et al.: FusionFS: toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In: Proceedings of the 2014 IEEE International Conference on Big Data (Big Data 2014). IEEE, Washington (2014)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

