DIESEL: A Dataset-Based Distributed Storage and Caching System for Large-Scale Deep Learning Training

Lipeng Wang
lwangay@cse.ust.hk
HKUST
Hong Kong, China

Songgao Ye
yesonggao@sensetime.com
SenseTime Research
Beijing, China

Baichen Yang
byangak@connect.ust.hk
HKUST
Hong Kong, China

Youyou Lu
luyouyou@tsinghua.edu.cn
Tsinghua University
Beijing, China

Hequan Zhang
zhanghequan@sensetime.com
SenseTime Research
Beijing, China

Shengen Yan
yanshengen@sensetime.com
SenseTime Research
Shenzhen, China

Qiong Luo
luo@cse.ust.hk
HKUST
Hong Kong, China

ABSTRACT

We observe three problems in existing storage and caching systems for deep-learning training (DLT) tasks: (1) accessing a dataset containing a large number of small files takes a long time, (2) global in-memory caching systems are vulnerable to node failures and slow to recover, and (3) repeatedly reading a dataset of files in shuffled orders is inefficient when the dataset is too large to be cached in memory. Therefore, we propose DIESEL, a dataset-based distributed storage and caching system for DLT tasks. Our approach is via a storage-caching system co-design. Firstly, since accessing small files is a metadata-intensive operation, DIESEL decouples the metadata processing from metadata storage, and introduces metadata snapshot mechanisms for each dataset. This approach speeds up metadata access significantly. Secondly, DIESEL deploys a task-grained distributed cache across the worker nodes of a DLT task. This way node failures are contained within each DLT task. Furthermore, the files are grouped into large chunks in storage, so the recovery time of the caching system is reduced greatly. Thirdly, DIESEL provides chunk-based shuffle so that the performance of random file access is improved without sacrificing training accuracy. Our experiments show that DIESEL achieves a linear speedup on metadata access, and outperforms an existing distributed caching system in both file caching and file reading. In real DLT tasks, DIESEL halves the data access time of an existing storage system, and reduces the training time by hours without changing any training code.

ICPP 2020, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404472

KEYWORDS

storage system, dataset management, deep learning, distributed cache, dataset shuffling

ACM Reference Format:
Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. 2020. DIESEL: A Dataset-Based Distributed Storage and Caching System for Large-Scale Deep Learning Training. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404397.3404472

1 INTRODUCTION

Deep learning training (DLT) tasks usually run on distributed DL frameworks, such as Caffe, PyTorch, and TensorFlow. Each task works on a dataset of a large number of small files with many categories and labels. For example, a popular image dataset for computer vision tasks, ImageNet-1K [8], contains more than 1.28 million files in 1,000 categories, and the Open Images [10] dataset contains around 9 million images. The average image sizes are 110KB and 60KB for ImageNet-1K and Open Images, respectively. Furthermore, training a deep learning model over a dataset requires many epochs, and in each epoch the files in the dataset are read in a randomly shuffled order. As such, file access in DLT tasks takes a significant amount of time.

Existing DL frameworks run on computer clusters and store datasets in shared file systems, e.g., Lustre[22]. On large computer clusters, the shared file systems are easily saturated with a large number of concurrent and random accesses on small files. Since accessing small files is a metadata-intensive workload, existing work stresses to improve the performance of the metadata server. Also, global caching systems, such as Quiver[9] and in-memory caching middleware, e .g., Memcached cluster, are deployed to relieve the I/O pressure on the underlying storage systems. However, these caching systems operates in file-level, which are slow to load datasets from underlying storage systems.

DLT任务存储和缓存有三个问题：1，访问包含大量小文件的数据集耗时较长；2，全局内存缓存系统容易受到节点故障的影响，恢复速度较慢；3，当数据集过于庞大无法完全缓存到内存时，反复以随机顺序读取件的效率低下。

DIESEL：
首先，由于访问小文件是一个元数据密集型操作，DIESEL 将元数据处理与元数据存储分离，并为每个数据集引入了元数据快照机制，从而显著加速了元数据访问。
其次，DIESEL 在 DLT 任务的工作节点上部署了一个基于任务粒度的分布式缓存系统，这样节点故障的影响被限制在每个 DLT 任务内部。此外，文件在存储中被分组为大块，从而大幅减少了缓存系统的恢复时间。
第三，DIESEL 提供了一种基于块的随机洗牌方法，在不牺牲训练精度的情况下，改进了随机文件访问的性能。

The I/O problem becomes even more severe when the computation is being sped up with powerful deep learning accelerators in clusters. For example, NVIDIA introduced the tensor core, which enables fast training with FP16 and INT8 instructions in the latest GPUs. Google's TPU (tensor processing unit) pods offer the ability to scale the training up to one thousand TPUs. With these massively parallel processors, the performance of data movements significantly falls behind the computation speed.

In this paper, we take advantage of some unique characteristics of file accesses in DLT to co-design DIESEL, a storage and caching system for DLT tasks. First, DLT tasks have separate read and write phases. Once a dataset is written to the cluster, it is read in multiple rounds without updates. This separation enables stateless metadata cache designs. We can store metadata in memory and use a mechanism to offload the computation of metadata to DLT worker nodes instead of on a metadata server. Second, in each training job, the dataset is read multiple times. This dataset-based access pattern motivates a distributed cache design for each dataset. Third, in DLT tasks, the input files are not required to be accessed in a fixed order, as long as the access order is randomized[9][35]. As such, without reducing the model accuracy, we can perform the data shuffling in a storage-friendly data layout with very small memory footprint.

In view of these characteristics, we first decouple the metadata processing and storage, leveraging the separate phases of write and read. In the writing phase, small files are compacted into chunks (e.g., 4MB) with metadata in an in-memory key-value database. To recover metadata quickly in case the metadata is lost in the in-memory store due to accidents (e.g., power failure), each chunk is self-contained with metadata encoded, and the chunks are organized in a sorted order. In the reading phase, since there is no update to the dataset, a metadata snapshot is distributed among DLT task nodes. Thus, the pressure of metadata processing is offloaded, and there is no centralized metadata server in this design. Secondly, we design a task-grained distributed caching system to aggregate free memory of the DLT tasks nodes and cache the training dataset. Compared with existing global in-memory caching systems, DIESEL contains the impact of node failures within individual DLT tasks. Since files are compacted into large chunks, the recovery time of our task-grained distributed caching system is also reduced significantly. Thirdly, we propose a chunk-wise shuffle method to generate random file orders in which the small file reads from the training framework are converted into chunk-wise reads to the underlying system, without sacrificing the model accuracy. As such, the read requests can be merged into large data chunk requests to utilize the I/O bandwidth effectively. The memory footprint of our chunk-wise shuffle method is small because we only need to cache a small fraction of the dataset.

All our designs are implemented in DIESEL. Evaluations show that DIESEL outperforms existing storage and caching systems, and significantly reduces the training time.

In summary, we make the following three contributions in this paper:

(1) We utilize the unique characteristics of DLT tasks to provide self-contained and fast recoverable writes, and support fast reads with metadata snapshots. The metadata access performance increases linearly with the number of clients in DIESEL.
(2) The task-grained distributed cache in DIESEL leverages the aggregate memory of the DLT task nodes and outperforms existing in-memory caching systems in both caching and reading speeds. It also contains the impact of node failures within individual DLT tasks.
(3) We propose a chunk-wise shuffle method to facilitate fast random file access order without sacrificing model accuracy. With the chunk-wise shuffle method enabled in DIESEL, the DLT tasks fully utilize the maximum read bandwidth when the dataset is too large to fit in the distributed cache.

2 BACKGROUND

2.1 I/O Characteristics of Deep Learning

For deep learning tasks, the very first step is to prepare the dataset, either downloading from public repositories or collecting private datasets (e.g., from cameras). The prepared dataset is then written to the distributed storage system in batches. This data preparation step provides us an opportunity to merge small files into large data chunks. For a dataset, the file-access characteristics in a training task are:

(1) *Separate write and read phases*: one training task only works on one dataset, and performs read-only access to the dataset.
(2) *Iterative and traversal read*: files are read epoch by epoch; in each epoch, all files are read once.
(3) *Shuffled file access order*: between epochs, files are shuffled to a different random order to avoid model overfitting. The model accuracy is unaffected no matter what random order the file access is in each epoch.

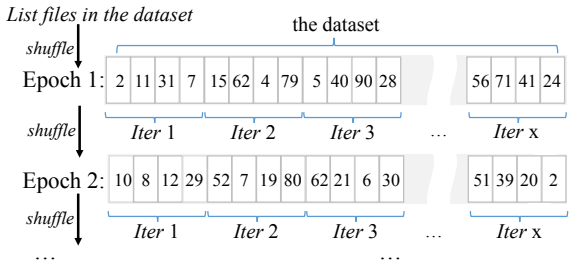


Figure 1: The dataset access pattern with a mini-batch (a fixed number of files) size of four: each epoch iterates all files in mini-batches in a random order. The numbers in the buckets represent file IDs.

Figure 1 illustrates the dataset access pattern in deep learning training tasks. At the beginning of a DLT task, all file names of a dataset are loaded into memory. Before each epoch, the training framework shuffles these file names to generate a random read order. The files are then read in mini-batches with iterations.

On a large computer cluster, many training tasks are running concurrently and each training task may work on a different dataset. These training tasks issue a very large number of small file read requests to the underlying distributed storage system. Such access patterns are challenging to the storage system and slow down the training processes due to the inefficiency of the storage system on processing small files. Therefore, we design and implement a

dataset-based storage and caching system to speed up these access patterns on training datasets.

2.2 Limitations of Current Systems

Accessing small files in a distributed storage system is a typical metadata-intensive workload. Existing storage systems on computer clusters (e.g., Lustre[22], Ceph[31], GlusterFS[17]) have multiple strategies to speed up the processing of metadata. For example, the latest versions of Lustre have a DNE (i.e., Distributed Namespace) feature to distribute either the sub-directories (DNE1) or the structure of a directory (DNE2) to multiple MDTs (metadata targets) to increase the performance on metadata operations. However, both DNE approaches have drawbacks. With DNE1, one sub-directory will be distributed to one MDT. If a very large number of files are in a directory, accessing the metadata on these files will issue requests to the same MDT server, which can cause saturation on that machine. With DNE2, the metadata of one directory will be striped to multiple MDTs. Then, accessing files in a directory (e.g., readdir) will traverse all stripes, which incurs significant overhead. Even given a highly efficient metadata design, DLT tasks, in which small files are read in random order cannot utilize the I/O bandwidth effectively.

Global in-memory caching systems such as a Memcached cluster are deployed to speed up file reads in clusters. However, without co-design with the storage system, a global in-memory caching system suffers from a number of drawbacks. For example, node failures may cause the global in-memory caching system to stop working; cache thrashing will occur when the global caching system cannot keep all active datasets in memory. Due to the random reads of DLT tasks, the reading performance drops a lot in such scenarios even if only a small part of datasets is inaccessible due to node failure or capacity limit. Additionally, the caching procedure itself takes a long time on clusters if files are small.

3 RELATED WORK

Optimization on Metadata Management Existing storage systems, such as TableFS[18], IndexFS[19] and DeltaFS[34][33], use the log-structured merge tree[14] (LSM-tree) to manage the metadata, and outperform local file systems on metadata-intensive workloads. GiraffaFS[24] stores the metadata in a high-performance distributed database named HBase[2]. HopsFS[13] replaces the single node in-memory metadata service in HDFS [23] with a distributed metadata service built on NewSQL database. Consequently, the capacity and throughput of metadata are both increased by tens of times over the original namenode. LocoFS [12] also uses key-value databases to manage the metadata. It decouples the directory content and structures into flattened records in different key-value databases. Metadata caching is extensively used in existing systems. For example, the clients of traditional file systems such as Lustre, GPFS[21] and Ceph, all have caching systems on the file namespace and attributes to speed up file lookup and directory traversal. Google GFS[4] stores metadata in main-memory on a single master node. In contrast, we use distributed key-value databases to store the metadata and offload the computation on metadata to DLT task nodes. Furthermore, with metadata snapshot enabled in DIESEL, all metadata operations will be served within each DLT process locally. This design can bypass the metadata server entirely and remove the bottleneck on metadata queries and network communication. Thus the metadata performance is improved significantly.

Distributed Storage/Caching for Deep Learning FanStore [32] is an MPI-based runtime storage system for deep learning. It partitions datasets into chunks and distributes chunks to multiple nodes. With a task-grained distributed cache, DIESEL provides all the functionalities of FanStore. Moreover, DIESEL interacts with the underlying storage system in large blocks and has a chunk-wise shuffle method to further improve DLT task performance in memory-constrained scenario. DeepIO [35] implements an in-memory storage system that is co-located with the training application. It uses memory-based shuffling and entropy-aware opportunistic ordering techniques to reduce small file reads from the backend storage. DLFS [36] extends DeepIO to NVMe-oF (non-volatile memory express over fabrics) devices. However, the user code must be modified to use their systems, e.g., call of API functions *seed*, *inner_read* and *bread*, in order to improve performance. In comparison, DIESEL provides a FUSE (filesystem in userspace) interface which can be mounted to a local folder. After mounting, the training framework, such as PyTorch and TensorFlow[1] frameworks, can read files as if it reads local disks. All training frameworks support such local file reading method out-of-box. In our *chunk-based shuffle* method (§4.3), after reading a file list from DIESEL, the training framework can access files as usual. Quiver[9] is a caching system that employs secure hash-based file lookup, co-ordinated eviction and benefit-aware techniques to improve the reading speed of DLT tasks. However, Quiver works on the caching layer, not in the storage layer. Existing systems miss opportunities of I/O optimization for DLT tasks due to a single focus on the caching or the storage layer. In contrast, DIESEL is a storage and caching co-designed system that supports both data writing and data reading. With multiple-level optimizations in metadata management, dataset caching and large chunk reads, DIESEL is highly effective for DLT tasks.

4 DIESEL DESIGN

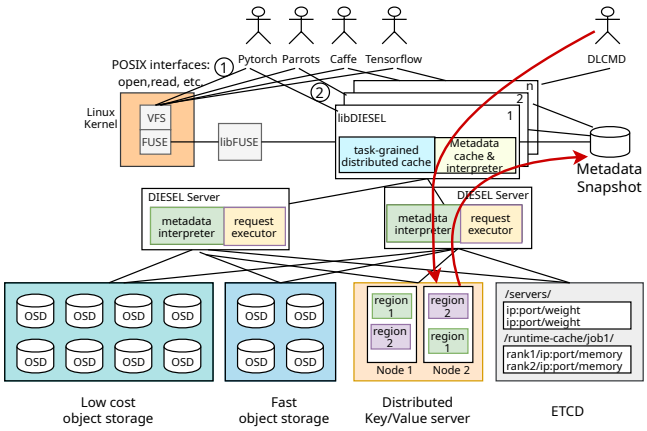


Figure 2: Overview of DIESEL

DIESEL improves the I/O performance for DLT tasks by introducing storage-caching co-design. Multiple optimization methods,

DIESEL中的request executor将小文件请求 sort + merge成chunk-wise operations

including data storage, metadata management, distributed caching and shuffling are integrated organically in DIESEL. Furthermore, simple but efficient interfaces are provided to users, DIESEL is mounted via the FUSE interface as local disks. Figure 2 illustrates the system architecture of DIESEL: the data chunks are stored in a shared object-storage (e.g., Ceph[31] or Lustre[22]); the metadata is stored in a distributed key-value database, e.g., Redis cluster; the system configurations are stored in an ETCD server. The DIESEL server hides the details of the underlying systems and provides a unified interface for the client to access data as well as metadata. It maintains the metadata in the key-value database and the data chunks in object-storage consistently. The request executor in the DIESEL server sorts and merges small file requests to chunk-wise operations. We provide two methods for the users to access files in their training code from DIESEL, mounting DIESEL to a local folder via FUSE [28] or reading files via the libDIESEL library. The FUSE-based filesystem is implemented atop the libDIESEL. The libDIESEL contains two main components: (1) The metadata cache and interpreter loads and interprets the metadata snapshot from local disk; (2) The task-grained dataset cache caches the training dataset in the main memory at runtime. A separate command-line tool (DLCMD, similar to s3cmd in Amazon S3) is provided to write and manage the datasets in DIESEL.

DIESEL server is deployed by the system admin on an existing object-storage and a distributed key-value database. Metadata snapshots of datasets can be automatically downloaded from the DIESEL server. When a DLT task starts, libDIESEL will load a metadata snapshot of the training dataset into local memory to serve subsequent metadata access.

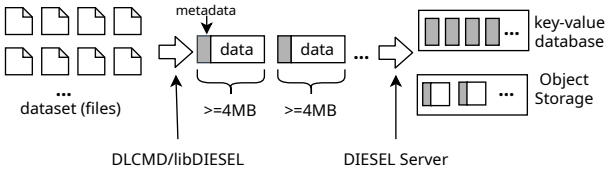


Figure 3: Write-flow from DIESEL client to DIESEL server.

Figure 3 shows the write flow of DIESEL. Users can use use libDIESEL or DLCMD to write files from local disk to DIESEL. To ease the I/O pressure on the shared storage system, files are aggregated into large data chunks ($\geq 4MB$) on the client-side. Metadata is stored in the head of data chunks. When the DIESEL server receives the data chunks from clients, it extracts the metadata to construct key-value pairs and writes them to the key-value database. The data chunks are sent to the underlying storage systems.

Figure 4 illustrates the flow of reading files in DIESEL. User applications issue the file read requests via the FUSE [28] interface or libDIESEL APIs. If the task-grained distributed cache is enabled on the client side, the file requests will be forwarded to corresponding peers that cache the requested files. If a cache miss occurs on the remote peer or the task-grained distributed cache is disabled, the file read requests will be sent to the server directly. If the server cache is enabled and the corresponding data chunks are cached in the fast object-storage (e.g., SSD-based), the file read requests will be sent to the fast object-store system. Otherwise, the slower

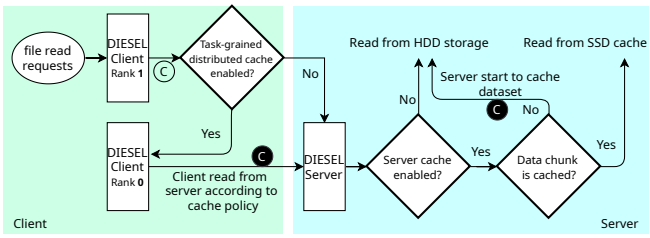


Figure 4: Read-flow from DIESEL client to DIESEL server.

object-storage (e.g., HDD-based storage) system will handle the requests. If a cache miss occurs on the server-side, the server will start to cache the dataset in the background. The "**C**" in a dark circle stands for data chunk operations. The caching stage of the task-grained distributed cache and the HDD-SSD server cache operates on data chunks with chunk size equal to or greater than 4MB. If the chunk-wise shuffle method is enabled, the read requests in a DIESEL client are performed on chunks (as shown as "**C**" in an unfilled circle in Figure 4).

4.1 Data Layout and Metadata Organization

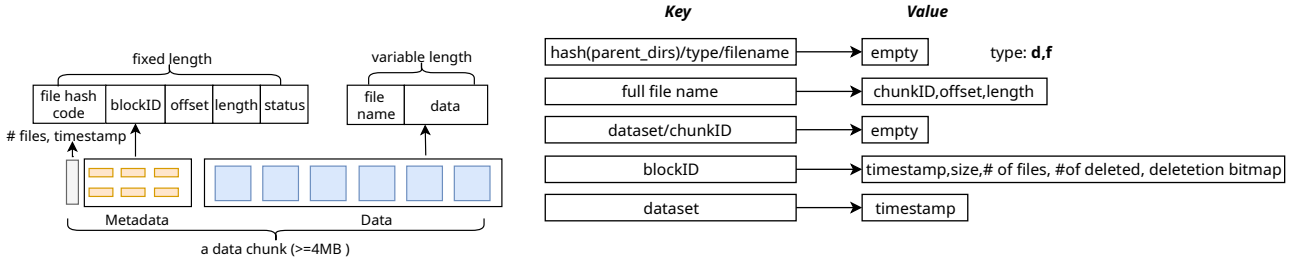
Figure 5 illustrates the data chunk layout and metadata design in our system. The data chunks are self-contained with metadata encoded in the header. The DIESEL server can rebuild all key-value pairs from data chunks. The key-value form metadata in the key-value database is shown in Figure 5b. When writing files into DIESEL, the server extracts file metadata from the head of data chunks and constructs these key-value pairs. With this design, DIESEL achieves excellent performance on metadata lookup.

4.1.1 Decoupling the Metadata Storage and Metadata Processing. Similar to other existing systems [12], the (de)serialization is performed in the DIESEL server instead of the key-value databases. File system operations on the metadata are translated to equivalent key-value operations in DIESEL servers. For example, listing files and directories (readdir) in a /folderA can be performed by using *'pscan hash(/folderA)/d ∪ pscan hash(/folderA)/f'* (pscan stands for scan with a prefix; 'd' and 'f' represents directory and regular file, respectively) in the key-value database. The metadata of an individual file and data chunk can be retrieved by a single *get* command in the key-value database. The metadata of a data chunk contains the update timestamp, size, number of files it contains, number of deleted files and the deletion bitmap. The folder hierarchy can be built dynamically from the full filenames in the key-value pairs. DIESEL supports modifying/deleting files by first deleting the old file and then writing a new file. There are house-keeping functions to merge chunks with holes caused by file modification and deletion.

Table 1: The composition of a data chunk ID.

Field	timestamp (in seconds)	machine identifier	process ID	chunk number counter
range (bytes)	0-3	4-9	10-12	13-15

4.1.2 Fault-Recovery on In-Memory Key-Value Metadata . This subsection discusses how we can recover the metadata when the main



(a) Data chunk layout in DIESEL **(b) Metadata design in key-value database**

Figure 5: Data chunk layout and metadata design in DIESEL.

memory-based key-value database crashes. There are two scenarios for an in-memory database such as in a Redis cluster:

(a) We lose some of recently written key-value pairs due to key-value server node failures.
(b) We lose all in-memory key-value pairs due to data-center power failure or other failures.

For scenario (a), we need to scan the data chunks from a certain timestamp whereas for scenario (b), all data chunks must be scanned in the order that they are written. To ensure data chunks are sortable by their update time, DIESEL generates chunk IDs based on timestamps. Table 1 shows the composition of a data chunk ID. The first four bytes are the timestamp (in seconds) of the creation time of this chunk. Following is six bytes of machine identifier (MAC address of the Ethernet interface) and three bytes of process ID in the system. An unsigned integer counter of three bytes at the end records the ID of the current chunk. This method allows each process to generate more than 16.7 million unique chunk IDs per second. The binary form of chunk ID is converted into printable characters (e.g., using base64 algorithm) and stored into the underlying object-storage. This way, the data chunks can be sorted by their IDs in their written order.

4.1.3 Metadata Snapshot. To further reduce the pressure on the metadata (key-value) database, DIESEL supports the materialization of the metadata snapshot of a dataset to local disk. The metadata snapshot is kept simple to reduce the download time and the snapshot size, containing the dataset update timestamp, the chunk ID lists and the file metadata (chunk ID, offset, length and full name). The file system hierarchy, which can be built from the file names, is constructed when a client loads the snapshot from disk. When libDIESEL loads a metadata snapshot from local disk, it compares the dataset name and the update timestamp to the records in the key-value database to check whether the metadata snapshot is up-to-date. Users need to download a new metadata snapshot if the timestamp does not match. The two arrows in Figure 2 shows the control path when users download a metadata snapshot from DIESEL. In a computer cluster, to reduce the operating cost of downloading metadata snapshots, users can save snapshots in a distributed file system (e.g., Lustre), where all nodes can access them concurrently.

4.2 Task-Grained Distributed Cache

A DLT task reads a dataset repeatedly, so caching the dataset in the client memory will increase the read performance significantly. Existing global in-memory caching systems aggregate free memory of all nodes of clusters to cache large datasets. However, due to the random read pattern of DLT tasks, the global in-memory caching systems will perform poorly in the presence of node failures. Figure 6 shows how the reading speed changes with different cache hit ratios in a DLT task. We launch a 20-node Memcached cluster and run 16 read clients on each node. Each client reads a random set of 128 files in each iteration. At iteration 30, we disable the Memcached instance on one node, and then at iteration 70, we disable the Memcached instance on a second node. This way when a file access goes to the disabled Memcached instance, a miss will occur and the file reading request will be redirected to the underlying Lustre filesystem. As shown in the Figure 6, the reading speed drops a lot even when only a small part of the dataset is inaccessible in the in-memory caching system (e.g., 5% cache misses reduces 90% reading speed). In DIESEL, to solve this problem, we employ a *task-grained distributed cache* to contain the failures of in-memory caching within each DLT task. Specifically, the dataset of a DLT task is cached among the DLT task nodes. Failures on these nodes will only terminate the computation of the affected DLT task. Since DIESEL stores datasets in large data chunks, the recovery procedure of the *task-grained distributed cache* effectively exploits the read bandwidth of the underlying storage system. In contrast, existing global in-memory caching systems read and cache datasets by individual files, which takes a long time to recover.

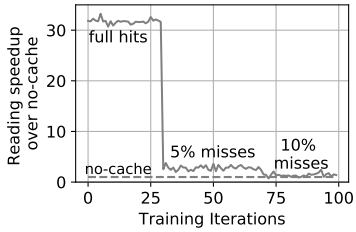


Figure 6: Reading speed changes with cache hit ratio in an in-memory caching system for DLT tasks.

Figure 7 illustrates the overview of this sub-system. Each computing node may run multiple I/O processes, for example, users may use four or more I/O workers to read files in PyTorch. When a training application starts, each I/O process will spawn a DIESEL client instance. If the dataset is distributed among all DIESEL client instances, each DIESEL client needs to create network connections

to all other peers in order to retrieve the entire dataset. This setup will result in a full mesh network with $n \times (n-1)$ connections (n is the total number of DIESEL clients). This number of network connections will be big in a typical training configuration with tens to hundreds of I/O workers and the large number of connections will cause significant memory and network overhead. To reduce the number of connections, DIESEL selects only one DIESEL client (the *master* client, which has the smallest rank) on each computing node to participate in dataset partitioning, and other clients will connect to the master client on each node to retrieve files. This way, the number of connections will be $p \times (n-1)$, where p is the number of physical nodes and n is the number of DIESEL clients ($p \leq n$ since one node will spawn one or more I/O workers in a typical training task). Compared with the multi-hop routing method used in DeltaFS[33], each DIESEL client can receive any file in the dataset by one hop, so the access latency is reduced. For example, the number of connections between clients is reduced by half (lines labeled 2) in Figure 7.

Cache Policies: Deep learning training tasks can start from loading previous training checkpoints (e.g., from an unfinished model before node failures) or pre-trained models to reduce the training time and improve the accuracy. To further improve the performance, DIESEL supports a oneshot cache policy: it pulls data from the server immediately after finishing registration to DIESEL servers. The DIESEL client caches the dataset in the background when the user loads the training models from disk. The read latency in the first epoch will be reduced this way. DIESEL also supports an on-demand cache policy, which pulls data chunks from DIESEL servers when a cache miss occurs. The on-demand cache policy has a little longer read latency in the first training epoch. After the first epoch finishes, all files in the dataset is accessed once from the backend storage and cached in clients' memory.

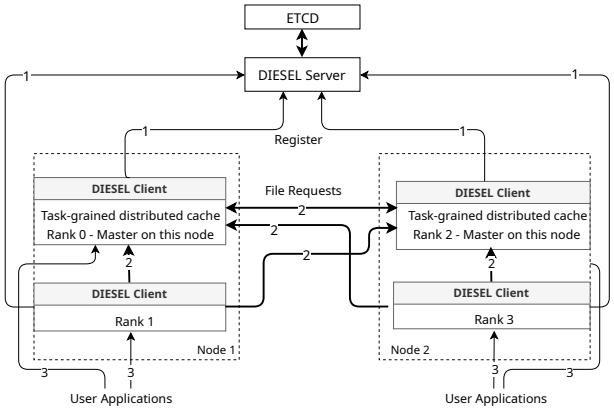


Figure 7: Overview of Task-grained Distributed Cache: lines labeled 1 stand for the registration requests of DIESEL clients; lines labeled 2 are network connections between DIESEL clients; lines labeled 3 represent the file read requests from user applications.

Table 2: The read bandwidth and IOPS with file size varied on an SSD-based storage cluster.

File Size(KB)	Bandwidth(MB)	Files/Second	4K-IOPS
1	33.54	34353.45	8588.36
4	128.28	32841.47	32841.47
16	464.44	29724.48	118897.92
64	1317.04	21072.64	337162.24
256	2725.93	10903.72	697838.08
1024	3104.26	3104.26	794690.56
4096	3197.68	799.42	818606.08

4.3 Shuffle in Memory-Constrained Scenario

DLT tasks usually read the dataset epoch by epoch with a randomly shuffled file order to prevent the model from overfitting. This random shuffle hurts the I/O performance a lot, especially when the dataset is too large to be cached in memory (as in Figure 6). The performance of traditional distributed storage systems also drops a lot with random small file reads. In contrast, with the increase of the read block size, the read bandwidth increases significantly (as shown in Table 2). Thus, DIESEL proposes a *chunk-wise shuffle* method to solve the following two problems:

(a) In memory-constrained scenario, the dataset cannot be fitted into the *task-grained distributed cache*, so how can DIESEL provide efficient reads?
(b) How to convert small file reads to large data chunk reads while keeping random shuffled file order in each training epoch?

Because the DLT tasks do not need a specific read order (as long as the order is different among epochs), and DIESEL has already stored datasets in chunks (as in Figure 3), we design the *chunk-wise shuffle* method, which converts small file reads to chunk-based reads automatically. The random read order is also maintained in the *chunk-wise shuffle* method.

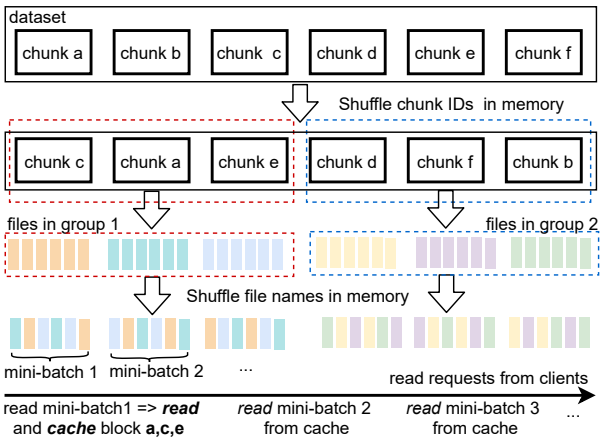


Figure 8: Chunk-wise shuffle method generates random file orders in which files can be converted to chunk-wise reads.

Figure 8 illustrates how our chunk-wise shuffle method generates a shuffled file list which can be converted into large chunk reads. This method works in three steps: Firstly, DIESEL shuffles data chunk IDs of the dataset. Secondly, the data chunks are split into

Table 3: libDIESEL API.

API	Parameter	Return Value	
DL_connect	user,key,dataset,server address and port	libDIESEL context (ctx)	connect to DIESEL server
DL_put	ctx, file, path in DIESEL	bool	put a file to DIESEL
DL_flush	ctx	bool	flush local buffer
DL_get	ctx, path in DIESEL	file content	get a file from DIESEL
DL_stat	ctx, path in DIESEL	file size, upload time, etc.	get a file's metadata from DIESEL
DL_delete	ctx, path in DIESEL	bool	delete a file in DIESEL
DL_ls	ctx, path in DIESEL	file and folder list	list files and folders in DIESEL
DL_save_meta	ctx, save path	bool	download datasets metadata to local disk
DL_load_meta	ctx, metadata file	bool	load datasets metadata from file
DL_shuffle	ctx	bool	enable chunk-wise-shuffle in DIESEL
DL_close	ctx	-	close libDIESEL context

multiple groups with a user defined group size. Thirdly, DIESEL extracts the files in each data chunk group and performs a random shuffle among these files. Finally, the final file list is generated by combining the file lists of all groups. When the training process reads files in the generated file order, DIESEL converts the file read requests to chunk-wise reads. Data chunks are read by the DIESEL clients and cached in memory. For file reads in each group, after the first few of mini-batch reads, subsequent file reads can be performed directly from DIESEL clients' cache. This way, the small file reads become large chunk-wise reads on the underlying storage systems to exploit I/O bandwidth. As shown in table 2, with 4MB size reads, the equivalent 4K-IOPS is about 25× greater than the 4KB reads. In addition, the memory requirement of the *chunk-wise shuffle* method is small compared to the original dataset size. Also, in real DLT tasks on the ImageNet-1K dataset, hundreds of data chunks (each chunk is around 4MB) in each group is sufficient to keep the accuracy of the training model.

5 IMPLEMENTATION

Our DIESEL server is implemented in around 5500 lines of Go code. Our DIESEL client is implemented in C++ and has around 20,000 lines of code. The DIESEL python library has around 2000 lines of python code.

DIESEL supports to use Ceph and other POSIX-based (e.g., Lustre) distributed storage systems to store the data chunks. When using Ceph, DIESEL works directly on the ceph-rados protocol (via *librados*). For metadata storage, DIESEL supports Redis cluster or other key-value databases. Users can choose the key-value database based on their system requirements, e.g., workload and hardware availability. We use Apache Thrift [26] as the RPC layer between the DIESEL server and client. Peers in the task-grained distributed caching system in DIESEL also use Thrift to exchange data.

User Interface. Both custom APIs and FUSE interfaces are provided for users to manipulate files in DIESEL. Generally, users use *DLCMD* (similar to the *s3cmd* in Amazon S3 storage) to store files into DIESEL. After that, the metadata snapshot can be downloaded from a DIESEL server to local disk. Users can also use the FUSE interface to retrieve files. Separate APIs are provided to users to manage the FUSE subsystem (i.e., mount, unmount). To support the generation of file list using the *chunk-wise shuffle* method in FUSE, DIESEL provides helper functions to let the user read the generated file list. The DIESEL code is highly optimized, for example, we use parallel-hashmap [16] to replace the standard hashmap in the STL. As FUSE redirection involves overhead [30], we use a multi-threaded loop in FUSE and employ multiple DIESEL clients within one FUSE mount to process concurrent POSIX reads for better performance.

Table 3 lists the APIs of the libDIESEL library. In addition, we provide housekeeping functions to manage datasets. For example, *DL_purge* is used to remove holes in data chunks caused by file modification and deletion; *DL_delete_dataset* is used to remove the entire dataset from DIESEL.

6 EVALUATION

In this section, we first evaluate the performance of DIESEL on file writing, since the first step in deep learning training is to load files into the system. Next, we evaluate the efficiency of metadata access and metadata snapshot. Then, we compare the performance of DIESEL's *task-grained distributed cache* with the popular main memory caching system *Memcached*. After that, we evaluate the *chunk-wise shuffle* method and compare the read performance of DIESEL with Lustre, a highly-optimized distributed file system. Finally, we test the read speed of DIESEL in real deep learning training tasks. Since DIESEL supports to retrieve files using custom APIs and POSIX interfaces (via FUSE), we denote the two alternatives as DIESEL-API and DIESEL-FUSE, respectively. We test the performance on 4KB and 128KB files in most tests since the sizes of most files in a dataset are in this range. For the evaluation of file operations (writing/reading), we use hundreds of millions of files with random contents to do the tests. Most of the tests on real-world dataset are conducted on ImageNet-1K [20] and CIFA-10 [7] datasets.

6.1 Experimental Setup

Table 4 shows the machine configurations for our experiments. We run DIESEL over Lustre, the most popular distributed file system in HPC clusters. The data chunks are stored in multi-level directories and the Lustre DNE feature is enabled to minimize the metadata overhead in Lustre. We have six storage machines running Lustre, each equipped with six 3.8TB NVME SSDs. Another ten machines run DIESEL and training applications. All nodes are connected by a 100Gbps Infiniband network. DIESEL is configured to use a Redis cluster as the metadata storage. We run a total of 16

Table 4: Server configurations.

	Storage Machine	Test Machine
# Machines	6	10
OS	CentOS 7.6 Kernel 3.10.0-957	
Network	100 Gbps Infiniband	
CPU	Intel(R) Xeon(R) Gold 6148 (20c40t) ×2	Intel(R) Xeon(R) Gold 6130 (16c32t) ×2
GPU	-	Tesla V100 SXM2 32GB ×8
Memory	512 GB	256 GB
Disk	NVME SSD 3.8T ×6	-
Software	Lustre 2.12.2	Memcached 1.4.15, FUSE 3.4.2, Twemproxy 0.4.1, Redis 5.0.5, Intel MPI 2017, PyTorch 1.1.0

Redis instances on four nodes, with four Redis instances on each node. We use our own MPI tool to execute file operations (writing/reading) concurrently on multiple nodes to simulate the I/O patterns of real DLT training frameworks. Specifically, we divide a list of file names evenly among MPI processes, and let each process write random contents and a hash code to the files. Then in the reading tests, each process reads files and checks the contents as well as the hash code for correctness. For deep learning training experiments, the official example code of PyTorch[15] is used without any code change, and these experiments run on four nodes, each of which is equipped with eight NVIDIA GPU cards. Because DIESEL implements an in-memory task-grained distributed cache, we compare its performance with *Memcached*, a popular in-memory object caching system. For the comparison to Memcached, since the original Memcached server does not support the cluster mode, we use *Twemproxy*[29] to build a Memcached cluster on the ten test machines. Each node runs a Memcached server with 16 threads and eight Twemproxy instances to provide a consistent hash[6] and unified namespace. We use the official *libMemcached* library to interact with the Memcached cluster.

6.2 Performance on File Writing

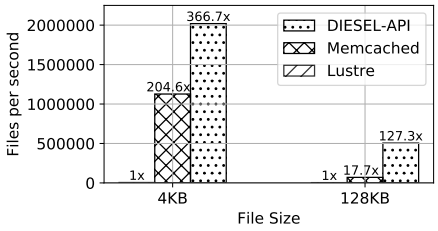
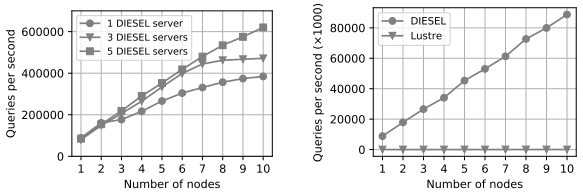


Figure 9: Performance comparison on writing files (4KB and 128KB).

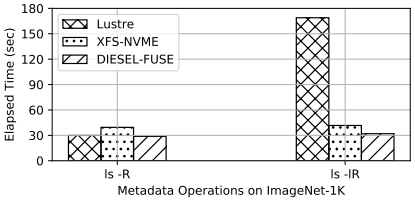
We first evaluate the performance of writing files into DIESEL, Lustre, and Memcached cluster. Figure 9 shows the comparison results on writing 4KB and 128KB files on 4 nodes with 64 MPI processes. DIESEL writes more than 2 million 4KB files per second, around 1.79× and 366.7× faster than Memcached and Lustre, respectively. This is because DIESEL clients aggregate files into large data chunks and then flush to DIESEL servers in batches. Compared to DIESEL, Lustre, as a general purpose file system, has complex file locks and metadata operations. Thus, the performance of writing small files in Lustre is much slower than DIESEL. The Memcached client library (libMemcached) does not have a batch mode for writing, although the twemproxy has pipelining support to merge requests from multiple clients into fewer requests to servers, in which each write request in the client must invoke a network RPC. The network overhead limits the performance of the Memcached cluster. Similar comparison results are shown on writing 128KB files. DIESEL is around 127.3× faster than Lustre and 17.3× faster than Memcached on 128KB writing. With 64 threads, DIESEL completes writing the ImageNet-1K dataset (around 1.28 million of files) from memory into the underlying storage system within only 3 seconds.

6.3 Efficiency of Metadata Access and Snapshot



(a) Metadata performance with 1,3 and 5 DIESEL servers

(b) Metadata performance with the metadata snapshot enabled



(c) Comparison of elapsed time on metadata operations

Figure 10: Performance on metadata operations.

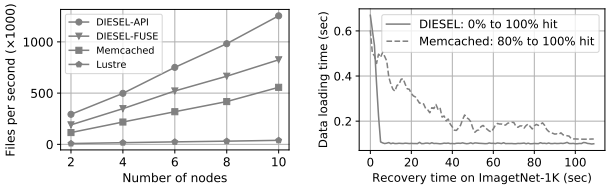
Metadata Access Efficiency. Figure 10 shows the results on metadata access with each node running 16 client threads. We first test the metadata performance by getting the file size in different configurations of DIESEL servers. Figure 10a show the QPS (queries per second) with the increase of the number of client nodes. The curves from bottom up represent 1 DIESEL server, 3 DIESEL servers and 5 DIESEL servers, respectively. With one DIESEL server, the QPS increase slows down starting from two client nodes. With three DIESEL servers, the QPS of metadata queries flattens with seven client nodes. This result shows that with the increase of number of DIESEL servers, the performance on metadata access scales better until it hits the maximum QPS of the key-value server, which is around 0.97 million/second in our Redis cluster configuration, tested by *memtier_benchmark* tool [11].

Metadata Snapshot Efficiency. When DIESEL clients load metadata snapshots into main memory, queries on metadata on these clients are accessed without contacting the metadata server subsequently. Figure 10b shows the metadata lookup performance increases linearly with the growth of number of clients. With one test node and ten test nodes, the QPS of metadata access is about

8.83 million and 88.77 million, respectively. The performance of metadata snapshot on ten nodes is around 1300× over the Lustre MDS server (around 68000 QPS in our test), in which all accesses have to request metadata from metadata servers.

We then test the metadata performance on the Lustre, XFS [27] (a high performance local filesystem) and the DIESEL-FUSE (with metadata snapshot enabled) on a single node by using the command-line tools (these commands are single-threaded). We use *ls -R* and *ls -lR* commands to test the elapsed time on ImageNet-1K dataset. The XFS filesystem runs on a 3.8TB NVME SSD. Figure 10c shows the results. The *ls -R* command gets the file names (i.e., readdir) only whereas the *ls -lR* command gets both the file names and sizes (i.e., stat). Lustre and DIESEL-FUSE take around 30-40 seconds to list all files in the ImageNet-1K folder. However, with the latter command, Lustre spends around 170 seconds to list file names and sizes. This long time is because Lustre stores the size information in the OSS (object-storage service) rather than MDS (metadata service, used to store file metadata), so getting a file size will involve multiple RPC calls in Lustre. Even with the LSOM (*lazy size on metadata*) option, in which Lustre stores approximate file size information in MDS, few POSIX clients utilize this feature in Lustre. In DIESEL, the file metadata is loaded from the local snapshot into main memory in hashmap. Therefore, the cost of getting the file metadata is $O(1)$ time complexity, from the implementation of the hashmap.

6.4 Efficiency of Task-Grained Distributed Cache



(a) Read performance comparison on 4KB files **(b) Data loading time with recovery time**

Figure 11: Performance evaluation on DIESEL with task-grained distributed cache.

For the evaluation of the task-grained distributed cache, we compare the random file read performance of DIESEL to the Memcached cluster since the task-grained distributed cache and the Memcached cluster both work in main memory. Figure 11a illustrates the performance results on 4KB file reading. With the increase of the number of clients, the performance of all four alternatives increases: DIESEL-API achieves more than 1.2 million QPS with 10 nodes. DIESEL-FUSE gets about 800, 000 QPS, which is more than 60% of the DIESEL-API is. The Memcached cluster is in the third place with around 560, 000 QPS. Lustre serves around 40, 000 files only, which is the slowest among all four systems. Because the dataset is split and cached among all Memcached servers, each client must have a network connection to each server. Therefore, the network overhead slows down the reading speed in the Memcached cluster. Compared to the DIESEL-API, the performance reduction in DIESEL-FUSE is due to the implementation of the FUSE sub-system in Linux: the kernel issues read-requests in small chunks concurrently with multiple threads and forwards these requests to the userspace process. Therefore, context switches cause non-negligible overhead when using the FUSE interface. With more features added to FUSE, the performance overhead will be reduced in the future.

Figure 11b compares data loading time and recovery time between DIESEL and the Memcached cluster on the ImageNet-1K dataset. We start the recovery from zero cache (0% hit ratio) to full cache (100% hit ratio) for DIESEL, and from 80% hit ratio to 100% for the Memcached cluster. The starting cache hit ratio of Memcached is set high because otherwise its data loading time will be excessively long. DIESEL tasks a very short time to load the entire dataset into the task-grained cache, but the Memcached cluster takes much longer in data loading. The data loading time for each batch read drops quickly and stabilizes at around 0.1 seconds in DIESEL within 10 seconds. In contrast, the Memcached cluster takes more than 100 seconds to finish caching all files even though it needs to load only 20% of the files to recover the cache of the dataset. The large chunk storage scheme helps DIESEL use less time to recover its in-memory caching system.

6.5 Efficiency of Chunk-Wise Shuffle Method

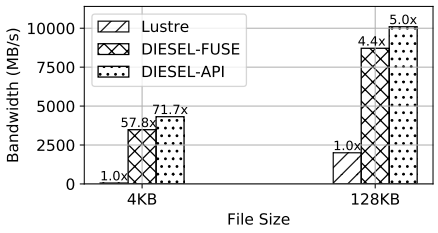
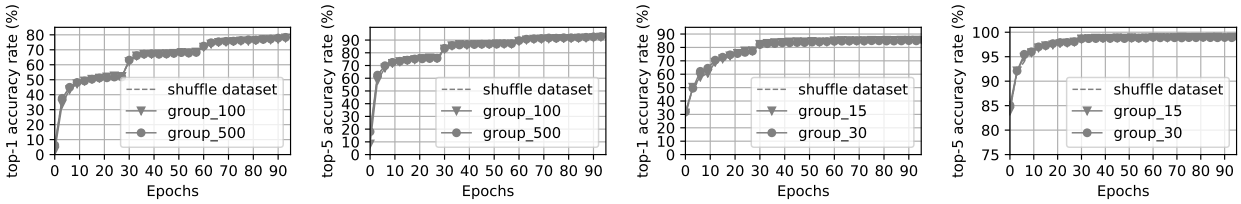


Figure 12: Comparison of read bandwidth on 10 nodes (160 threads).

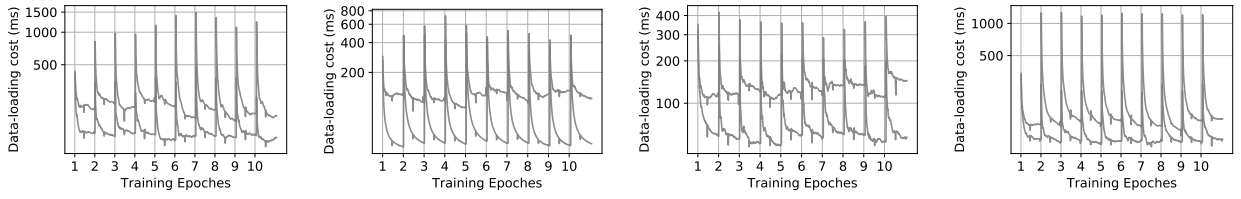
In this section, we evaluate the efficiency of the chunk-wise shuffle method on the read bandwidth, as well as the accuracy and convergence speed on model training. Figure 12 shows the read bandwidth of DIESEL and Lustre on 4KB and 128KB files on ten nodes with 16 threads on each node. Lustre gets around 60.2MB/s (or 15411 files/second) on 4KB reads whereas DIESEL-API and DIESEL-FUSE achieve around 4317MB/s (or over 1.1 million files/second, 71.7× over Lustre) and 3483.7MB/s (or around 0.89 million files/second, 57.8× over Lustre), respectively. In 128KB reads, Lustre gets about 2001.8MB/s bandwidth. DIESEL-API reaches around 10095.3MB/s, and DIESEL-FUSE reaches around 8712.5MB/s. The read bandwidth of DIESEL-API is 5.0× faster than the Lustre whereas the DIESEL-FUSE is 4.4× faster than the Lustre on reading 128KB file. The chunk-wise shuffle method improves the read bandwidth significantly, especially for 4KB files.

Figure 13 illustrates the model accuracy and convergence speed of two models: ResNet-50 on the ImageNet-1K dataset (Figure 13a and 13b) and ResNet-18 on the CIFAR-10 dataset (Figure 13c and 13d). The top-1 accuracy examines the percentage of the model answer exactly matching the correct answer whereas the top-5 accuracy represents the percentage of any of the top five answers



(a) Top-1 accuracy on ImageNet-1K/ResNet-50 (b) Top-5 accuracy on ImageNet-1K/ResNet-50 (c) Top-1 accuracy on CIFAR-10/ResNet-18 (d) Top-5 accuracy on CIFAR-10/ResNet-18

Figure 13: Top-1 and top-5 accuracy over epoch on two models.



(a) AlexNet (b) VGG-11 (c) ResNet-18 (d) ResNet-50

Figure 14: Data access time in each iteration on four models (the top curve is Lustre, and the bottom curve is DIESEL-FUSE).

matching the correct answer. We use different chunk group sizes in the tests: on the ImageNet-1K dataset, the group size is set to 100 and 500 since the dataset is large enough whereas on the CIFAR-10 dataset, the group size is set to 15 and 30 since the dataset size is relatively small. Compared to the original shuffle-over-dataset (donates as *shuffle dataset* in Figure 13) method, our chunk-wise shuffle method affects neither the model accuracy nor convergence speed. The memory consumption of the chunk-wise shuffle method on the ImageNet-1K dataset (around 150GB) is around 2GB only. In other words, in memory-constrained scenarios, by caching and shuffling files in the chunk-wise shuffle method, the reading speed achieves over 88.12% of the task-grained distributed cache (the fully cached scenario).

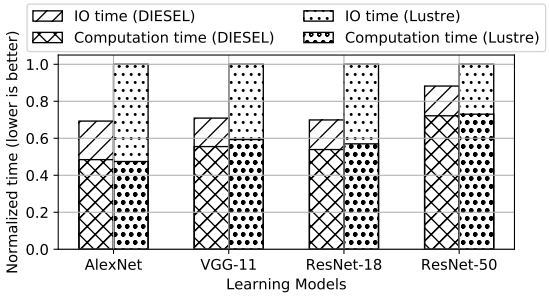


Figure 15: Training time comparison between DIESEL-FUSE and Lustre on four models.

6.6 Evaluation in DLT Tasks

In this section, we compare the training time in real deep learning training tasks between Lustre and DIESEL-FUSE. Since users prefer using the standard POSIX interfaces to read files (e.g., deep learning training frameworks such as PyTorch have built-in dataloaders that work on POSIX interfaces), DIESEL-FUSE is user-friendly for most deep learning scientists.

We evaluate the data access time on four models (i.e., AlexNet[8], VGG-11[25], ResNet-18[5] and ResNet-50[5]) on the ImageNet-1K dataset. We use the official PyTorch example [3] code on four nodes with a total of 32 GPU cards. The data access time includes data shuffling time and reading time from the data source to the main memory. The data access and computation is already pipelined (i.e., there are separate I/O threads to read files while the GPU compute gradients) in the training framework. Figure 14 illustrates the average data access time of the first ten epochs of the four models. There is a shuffle stage to generate random file orders in each epoch. Thus the average data access time goes up in the first iteration of each epoch. For all the models, the data access time of DIESEL-FUSE is about half of the data access time of the Lustre. Training the well-known ResNet-50 model on the ImageNet-1K dataset usually takes more than 90 epochs. Each epoch has 5005 iterations with a typical mini-batch size of 256. DIESEL-FUSE saves 80 milliseconds for each iteration in our tests, so reading files from DIESEL-FUSE can save around 35946 seconds, which is about 10 hours.

Figure 15 shows the normalized total training time comparison between DIESEL-FUSE and Lustre on four models (time is normalized to the total training time on the Lustre) on four test machines. The total training time of these four models range from 37 to 66 hours on Lustre on the four Pytorch workloads. DIESEL-FUSE reduces the IO time by 51-58%, and consequently the total time by 15-27% to 29 to 57 hours, respectively. This 8-9 hour performance improvement is expected to be similar on other training frameworks (e.g., TensorFlow), because the amount of I/O access is unchanged. With smaller and lightweight models and more powerful accelerators, the data loading time will be a significant part of DLT tasks. DIESEL saves data loading time and improves the overall training time.

7 CONCLUSION

In this paper, we design and implement DIESEL, a customized storage and caching system for DLT tasks. We introduce the storage-caching co-design of DIESEL with many optimizations at the level of dataset. For metadata, we separate the metadata storage and processing, and implement a metadata snapshot method. This method bypasses the metadata server and achieves linear speedups in metadata access. A task-grained distributed caching system is introduced in DIESEL, which contains the impact of node failure within individual DLT tasks. This caching system outperforms an existing global in-memory caching system in both file access and caching. Furthermore, we implement an efficient method to convert the random small file reads to chunk-wise reads. This method exploits the I/O bandwidth of the storage system in memory-constrained settings. In evaluations in real DLT tasks, DIESEL reduces the data loading time and consequently the overall training time significantly.

ACKNOWLEDGMENTS

This work was supported by the research project No. 18191980 sponsored by SenseTime Group Limited.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[3] Facebook. 2019. *Pytorch-example*. https://github.com/pytorch/examples/tree/master/imagenet

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[6] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, Vol. 97. 654–663.

[7] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[9] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An Informed Storage Cache for Deep Learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 283–296.

[10] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, and Vittorio Ferrari. 2018. The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale. *arXiv:1811.00982* (2018).

[11] Redis Labs. 2019. *NoSQL Redis and Memcache traffic generation and benchmarking tool*. https://github.com/RedisLabs/memtier_benchmark

[12] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. 2017. Locofs: A loosely-coupled metadata service for distributed file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 4.

[13] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. 2017. Hopsfs: Scaling hierarchical file system metadata using newsql databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 89–104.

[14] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[15] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS-W*.

[16] Gregory Popovitch. 2019. *A header-only, very fast and memory-friendly hash map*. https://github.com/greg7mdp/parallel-hashmap

[17] Red-Hat. 2019. *Gluster Filesystem*. https://www.gluster.org/

[18] Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 145–156.

[19] Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 237–248.

[20] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[21] Frank B Schmuck and Roger L Haskin. 2002. GPFS: A Shared-Disk File System for Large Computing Clusters.. In *FAST*, Vol. 2.

[22] Philip Schwan et al. 2003. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, Vol. 2003. 380–386.

[23] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. 2010. The hadoop distributed file system.. In *MSST*, Vol. 10. 1–10.

[24] Konstantin V Shvachko. 2016. *Giraffa: A distributed highly available file system*. https://github.com/GiraffaFS/giraffa

[25] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[26] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook White Paper* 5, 8 (2007).

[27] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System.. In *USENIX Annual Technical Conference*, Vol. 15.

[28] Miklos Szeredi and Nikolaus Rath. 2019. *Fuse: Filesystem in Userspace*. https://github.com/libfuse/libfuse

[29] Twitter. 2019. *A fast, light-weight proxy for memcached and redis*. https://github.com/twitter/twemproxy

[30] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 59–72.

[31] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 307–320.

[32] Zhao Zhang, Lei Huang, Uri Manor, Linjing Fang, Gabriele Merlo, Craig Michoski, John Cazes, and Niall Gaffney. 2018. FanStore: Enabling Efficient and Scalable I/O for Distributed Deep Learning. *CoRR* abs/1809.10799 (2018). arXiv:1809.10799 http://arxiv.org/abs/1809.10799

[33] Qing Zheng, Charles D Cranor, Danhao Guo, Gregory R Ganger, George Amvrosiadis, Garth A Gibson, Bradley W Settlemyer, Gary Grider, and Fan Guo. 2018. Scaling embedded in-situ indexing with deltaFS. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 30–44.

[34] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. 2015. DeltaFS: Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop*. ACM, 1–6.

[35] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. 2018. Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 145–156.

[36] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam Moody, and Fahim Chowdhury. 2019. Efficient User-Level Storage Disaggregation for Deep Learning. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–12.