2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-7652-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/HPCA56546.2023.10070964

iCACHE: An Importance-Sampling-Informed Cache for Accelerating I/O-Bound DNN Model Training

Weijian Chen, Shuibing He*, Yaowen Xu, Xuechen Zhang†,
Siling Yang, Shuang Hu, Xian-He Sun‡, and Gang Chen
Zhejiang University †*Washington State University Vancouver* ‡*Illinois Institute of Technology*
**Corresponding Author: Shuibing He (heshuibing@zju.edu.cn)*

***Abstract*—Fetching a large amount of DNN training data from storage systems incurs long I/O latency and fetch stalls of GPUs. Importance sampling in DNN training can reduce the amount of data computing on GPUs while maintaining a similar model accuracy. However, existing DNN training frameworks do not have a cache layer that reduces the number of data fetches and manages cached items according to sample importance, resulting in unnecessary data fetches, poor cache hit ratios, and random I/Os when importance sampling is used.**

In this paper, we design a new importance-sampling-informed cache, namely, iCACHE, to accelerate I/O bound DNN training jobs. iCACHE only fetches parts of samples instead of all samples in the dataset. The cache is partitioned into two regions: H-cache and L-cache, which store samples of high importance and low importance respectively. Rather than using recency or frequency, we manage data items in H-cache according to their corresponding sample importance. When there is a cache miss in L-cache, we use sample substitutability and dynamic packaging to improve the cache hit ratio and reduce the number of random I/Os. When multiple concurrent jobs access the same datasets in H-cache, we design a model to assign the relative importance values to cached samples to avoid cache thrashing, which may happen when there is no coordination among the concurrent training jobs. Our experimental results show that iCACHE has a negligible impact on training accuracy and speeds up the DNN training time by up to 2.0× compared to the state-of-the-art caching systems.

I. INTRODUCTION

Deep neural networks (DNNs) have been attracting attention in computer vision [41], natural language processing [31], robotics [37], and many other fields. DNN training often needs to fetch data from I/O systems and compute them for updating parameters [4], [5]. Recent research shows that I/O has become the bottleneck in DNN training [27], [36]. This is because AI accelerators, such as GPUs and ASICs, have evolved at a faster pace than storage devices. Another reason is that DNN model training needs to access ever-increasing datasets. For example, the Google OpenImages dataset used in the Open Images Challenge is about 18 TB [12]. And training data items are shuffled every epoch to ensure that they are accessed in a random order, leading to poor I/O efficiency of storage systems.

Importance sampling (IS) [18], [20], [23], [24] is an approach to accelerate DNN training by skipping the calculation of some items while maintaining a similar accuracy. It assigns each data item an importance value to reflect its influence on DNN model accuracy. We refer to data items of high and low importance values as H-samples and L-samples respectively. When importance sampling is used, H-samples are computed in a higher probability while L-samples are computed in a lower probability. However, all existing sampling approaches are *computing-oriented IS (CIS*) algorithms because they only focus on reducing computing on GPUs instead of I/O. They still need to fetch all items to memory and thus perform poorly for I/O-bound DNN Training. For example, when training ResNet18 with CIFAR10 on a parallel file system, a history-based CIS algorithm [18] can only accelerate the overall training by 1.02× though the computing time is reduced by 1.3× (Section II-B).

Data caching is another widely used method to accelerate DNN training [27], [36], [53]. By caching reused samples in the fast memory of training servers, it avoids a number of slow I/O accesses from storage. However, existing cache algorithms for deep learning need to serve all fetches from data loaders, losing the opportunity to optimize I/O performance by serving fewer fetches with negligible accuracy degradation. Moreover, they do not consider sample importance. For example, CoorDL [36] never replaces data items in its MinIO cache. Therefore, it is possible that the MinIO cache does not have space for H-samples after it is full. Quiver [27] exploits substitutability to avoid memory thrashing. However, it is likely H-samples are substituted by L-samples leading to poor accuracy of DNN models after training.

None of the existing approaches can reduce the amount of data fetched and consider sample importance in the I/O of DNN training. In this paper, we propose the idea of *I/O-oriented importance sampling (IIS)* and apply it to data caching. IIS only fetches a subset of samples instead of all the original samples from the cache or storage. Simply caching H-samples for DNN training does not work well in the context of importance sampling. The existing cache replacement algorithms are designed to explore temporal locality based on recency or frequency. However, importance sampling accesses data items randomly and based on their impact on the model accuracy. Therefore, we need a new importance-sampling-informed cache replacement algorithm to achieve a higher hit ratio in the cache.

There are three challenges in the design of the new cache system. First, the importance values of data items may change across epochs. We need an efficient algorithm to keep a maximum number of H-samples in the cache when the importance

978-1-6654-7652-2/23/$31.00 ©2023 IEEE

values of samples are changed. Second, caches have limited space. If we only cache H-samples, data loaders still need to access L-samples randomly with poor I/O efficiency. Third, caches needs to serve multiple jobs. We can only achieve sub-optimal performance if there is no coordination between jobs.

To address these issues, we design and implement iCACHE, a new importance-sampling-informed cache software to accelerate DNN training when I/O is its performance bottleneck. Specifically, iCACHE is partitioned into two regions: H-cache and L-cache which store H-samples and L-samples respectively. We use a small-top heap (H-heap) for cache management. When H-cache is full, the data item corresponding to the node at the top of the heap will be evicted if its importance value is smaller than that of the incoming one. To efficiently refill the cache when importance values are changed, we manage a shadow heap for H-heap. After the importance values are updated, the H-heap becomes read-only and is used only for item eviction from the cache. The changes (i.e., insertions/evictions and value updates) to the H-heap are recorded in the shadow heap. To reduce the amount of random I/Os for L-samples, iCACHE uses dynamic packaging to load L-samples to L-cache in batch. When L-samples to be accessed are not in the L-cache, we apply substitutability to replace the missing L-samples with those already in the cache, thus reducing the number of small random I/Os and keeping a high training accuracy. When multiple jobs accessing the same dataset are serviced by iCACHE, for cache management, we design an approach to evaluate the cost-effectiveness of caching for each job. Then we recompute relative importance values for data items given all the values from the jobs that are deemed to benefit from the cache. iCACHE uses the relative importance value for cache management to achieve job coordination.

In summary, this paper offers the following contributions:

- We propose the idea of I/O-oriented importance sampling (IIS) and integrate it into the cache system, iCACHE, which can mitigate the I/O bottleneck in DNN training by reducing the number of samples to be fetched for each epoch training.
- We design a cache replacement algorithm based on sample importance, a dynamic packaging technique, and a multi-job handling mechanism to further boost the I/O system performance.
- We implement iCACHE in PyTorch [40] and evaluate it with eight DNN models on two datasets. The evaluation shows that iCACHE outperforms the state-of-the-art DNN cache systems Quiver [27] and CoorDL [36] by up to 2.0× and 1.9× on the model training time and 2.3× and 3.9× on the I/O time while achieving an equivalent training accuracy. The codebase of iCACHE is available at https://github.com/ISCS-ZJU/iCache.

II. Background and Motivation

A. I/O-Bound DNN Model Training

DNN training is an iterative process and the model accuracy converges gradually. All training samples in datasets are required to be read multiple times. It is called one *epoch* when all training samples are accessed exactly once. Each epoch consists of multiple *iterations*. Each iteration trains with a mini *batch* of data items, and the batch size is set by the user.

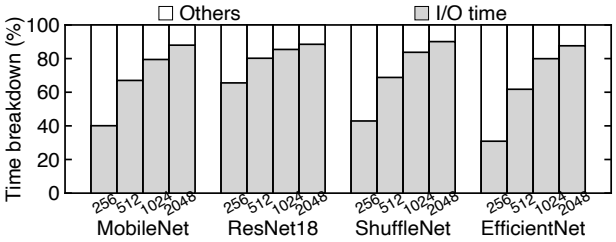


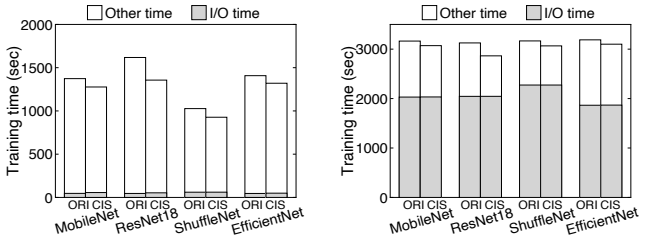
Fig. 1. Training time with varying batch sizes on four A100 GPUs.

Because data sizes of input training datasets are increasing at a rapid speed in DNN model training [27], in this paper, we assume all training data cannot be stored in local storage systems and they need to be read from a remote storage server (or a parallel file system). This is a typical configuration in modern HPC-AI data centers [9], [28], [39]. I/O performance is affected by network bandwidth, which is shared among multiple jobs accessing the server simultaneously.

In training, DNN framework needs to (1) load a mini batch of data items from a remote storage system to host memory; (2) pre-process the data items (e.g., rotation, cropping, and deformation) using CPUs; and (3) train the DNN model using the pre-processed data and GPUs. In each epoch, data loaders need to feed all samples in a random order to the training process with equal probability and exactly once. This is because global randomness is essential to guarantee model accuracy. As a result, loading a mini batch produces a large number of random reads issued to the back-end storage systems. When the training process is blocked by data loaders, it is called data stall. In this scenario, the training process is I/O bound and GPU and CPU utilization will be low.

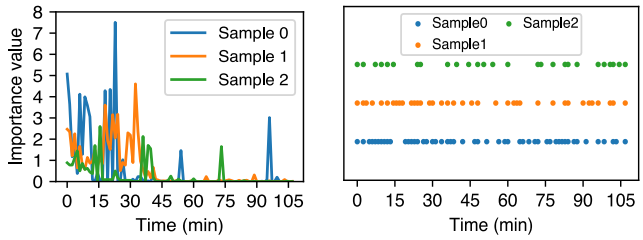
Although data prefetching, data caching, batch size adjustment, and multi-GPU training are widely used to accelerate DNN training, they are inefficient for I/O-bound tasks for the following reasons. (1) Prefetching is effective only when computing time is longer than I/O time. With powerful GPUs like A100 and H100, the computing time can be less than I/O time [36]. (2) Caching policies are usually based on traditional temporal/spatial locality, thus they are insufficient for DNN workloads with strong randomness [27]. (3) Batch size adjustment and multi-GPU training are mainly used to boost computing performance instead of I/O time.

To verify this, we train four DNN models on a server with four A100 GPUs with various batch sizes. The dataset is CIFAR10 and placed in a remote OrangeFS file system. The system configuration is described in Section V-A. We enable the built-in prefetching technique in PyTorch and implement an LRU-based cache system. The cache size is 20% of the training dataset, as Quiver [27] does. Figure 1 shows that even with existing performance optimizing techniques, I/O is still a bottleneck for multi-GPU training cases. For example, the I/O bottleneck becomes more prominent because the ratio of the



(a) Computing-bound training. (b) I/O-bound training.

Fig. 2. Training time with and without the historical-value based CIS algorithm in computing. "ORI" means the original training system without CIS.



(a) Varying importance values. (b) Sample selection during training.

Fig. 3. The varying importance values and the selected samples by the importance sampling algorithm during training. One dot in (b) means the sample is selected once.

I/O time to the total training time increases from 44% to 89% on average while the batch size increases from 256 to 2048.

B. Computing-Oriented IS Approaches are Inefficient for I/O Bound Training

Importance sampling accelerates DNN model training by feeding fewer samples on GPUs for updating parameters. When it is applied in model training, the random order of sample computed may be changed to reflect the principle that H-samples should be computed more frequently than L-samples. Recently, much research focuses on finding importance criteria to estimate the importance of samples, including model loss [18], [32], last-layer gradient [20], self-defined upper bound [24], or training another light model [49].

However, all existing IS algorithms are originally designed to only reduce the computing time of unimportant samples for computing-bound tasks, where I/O is not a performance bottleneck; they still load all samples from cache or storage. Such computing-oriented IS (CIS) approaches are efficient for computing-bound training, but perform poorly for I/O bound tasks because they cannot reduce the I/O time of the training process. To validate this, we measure the training time of the above four popular DNN models on CIFAR10 with and without a historical-value-based CIS algorithm [18]. We use a single A100 GPU for training and set the batch size to the default 256. We also enable PyTorch's prefetching technique. As Figure 2(a) shows, when CIFAR10 is placed in a local DRAM-based tmpfs (without a cache), CIS can reduce the computing time and the total training time by 1.3× and 1.2×. However, when we use a LRU-based cache that holds 20% of the training data and place CIFAR10 in a remote OrangeFS, CIS can only reduce the total training time by 1.02×, as shown in Figure 2(b). This is because CIS only shortens the computing time but I/O is the bottleneck in the latter case. The I/O bottleneck is caused by many factors, including dataset size exceeding local cache, low IOPS from remote storage nodes, data loading requiring considerable host CPU resources, and so on.

Inspired by the idea of CIS, it is feasible to apply I/O-oriented importance sampling (IIS) to fetch fewer samples from the cache or storage to accelerate I/O bound DNN model training with acceptable accuracy degradation.

C. Importance-Sampling-Informed Cache and Challenges

None of the existing cache systems are designed for DNN model training based on importance sampling. OS page cache explores temporal locality and uses recency or frequency for managing the data items when it is full [19], [34]. When data items are randomly accessed and the time of revisiting the same data items in the page cache is long, the page cache will have a high miss ratio. CoorDL [36] keeps all data samples in the cache with no eviction to avoid thrashing. The cache is used to store both H-samples and L-samples. When it is full, H-samples will not be stored in the cache, leading to a higher miss ratio of H-samples, which are accessed more frequently than L-samples. When Quiver [27] is used, an H-sample which is not in the cache may be substituted by an L-sample, reducing the model accuracy.

There are three challenges in the design of the importance-sampling-informed cache.

Varying importance values. The importance value of one sample changes across epochs during training [18], [32]. To verify this, we record the importance values of three samples (i.e., Sample 0 to Sample 2) with a model loss-based importance sampling algorithm [18] when training ResNet18 on CIFAR10. As Figure 3 shows, the same sample is selected from time to time with varying importance values. The importance value of the same sample changes because it is determined by the sample content and the model's paremeters (e.g., weights) which are updated by the SGD algorithm iteratively [7]. Therefore, the H-samples in the previous epoch may become L-samples. Since samples are selected according to their relative importance values, it is not practical to set an importance threshold to decide whether to place a sample in the cache or not. We need a judicious cache management algorithm to keep a maximum number of H-samples in the cache without significantly affecting model accuracy when the importance values of samples are changed.

Random I/Os after cache misses. A cache has limited capacity. It cannot always store all H-samples. When cache misses happen, it is required to read H-samples randomly. Furthermore, although H-samples are accessed more frequently than L-samples, training frameworks still need to access L-samples to improve sample diversity for high training accuracy. The performance of loading L-samples from storage systems may become the I/O bottleneck causing data stalls. A

widely used method of mitigating this problem is packaging the training dataset into many large files, each of which contains a certain number of L-samples [2]. However, it is not practical in this case because importance sampling specifies the order of samples to be trained. They are probably distributed in different data packages and will cause a serious read amplification problem if we directly use the existing packaging algorithms.

Cache misses caused by no job coordination. Multiple jobs accessing the same dataset is a common scenario in DNN training with different hyper-parameters [27], [36] or different models [27]. To effectively use the cache space, these jobs may share the cache. In this scenario, a training sample will receive different importance values from different jobs. Thus, it is difficult to decide whether the sample should be cached/replaced if we only consider its own importance values. When there is no coordination between the jobs, an evicted sample may be immediately fetched again from the storage system by another job, resulting in memory thrashing.

III. Design of iCache

In this section, we present the design of iCache. We first introduce the system overview of iCache and then elaborate on its three key components.

A. System Overview

iCache is an intelligent cache system for accelerating DNN model training. It supports both single-node with multiple GPUs training and multi-node distributed training, both of which are popular deep learning training configurations [2], [3], [27], [36], [44]. To illustrate the details, we first show the single-node architecture in Figure 4. It consists of client modules, cache managers, and servers.

iCache client and server. The client modules are integrated into the deep learning frameworks (e.g., PyTorch and TensorFlow). It mainly plays the role of request forwarding. When data loaders of DNN applications start to randomly select samples to read, the clients will forward the request to iCache servers by calling the RPC interface. One client belongs to a unique DNN training job and is transparent to the users.

A client module maintains an H-list to record H-samples for the training job. H-list is generated by the importance sampling algorithm. It is a list of vectors $< ID, IV >$, where ID corresponds to a sample's identity and IV is its importance value. Both the ID and IV are 64 bits (8B), thus the space overhead of H-list is trivial. We take a cache for ImageNet-1K (with 1281167 samples in 140GB) as an example. Assume the cache holds 20% samples, then the cached data size is 140GB*20%=28GB, and the importance mapping overhead is 1281167*20%*16B=3.9MB, which is just 0.014% (3.9MB/(28GB+3.9MB)) of the whole cache space. The IVs are computed according to the importance sampling algorithm. Although many algorithms exist, we choose the loss-based importance sampling algorithm [18] in our current design for its simplicity and efficiency. We wish to study other algorithms in future work. Since the importance value of one sample changes across epochs during training, we periodically updates importance values (Section III-B).

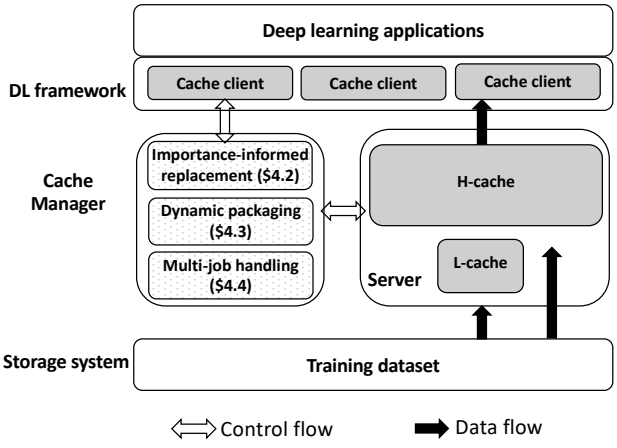


Fig. 4. The single-node architecture overview of iCache.

As mentioned in Section II-B, existing CIS algorithms can not reduce I/O time because they still need to load *all samples* from the storage or cache system. Therefore, we propose I/O-oriented IS (IIS) to reduce the number of fetched items in each epoch. Specifically, IIS decides the samples to be trained for the current epoch based on their historical importance values before the start of each epoch, then fetches and trains them. Thus, samples that have not been selected do not need to be loaded. In this way, both I/O and computing time can be reduced. At the end of each iteration, the host reads the losses of mini-batch samples from GPU memory and uses them to update the importance values of mini-batch samples. After each epoch, if a sample is selected and trained on GPUs, the host will get its updated importance value regardless of whether it is in cache or not. Otherwise, its importance value will be unchanged. The host then builds the H-list based on the updated importance values and uses the H-list to guide the DNN training in the next epoch.

The functionality of the iCache server is to provide a user-level cache, which stores training datasets in memory to accelerate the I/O-bound DNN training.

Cache manager. It is designed to manage the cache based on the importance values of samples. Since the importance values referenced by the sampling process are periodically updated by the historical-value based IS algorithm, the cache manager periodically pulls the H-list from clients to achieve a decent trade-off between cache performance and its management overhead. Additionally, the manager needs to dynamically pack samples which are later loaded to L-Cache. The minimum I/O unit is a package of samples.

Finally, when multiple jobs access the same dataset, the manager needs to determine how to manage data items based on multiple importance values corresponding to the same sample but from different jobs and the cost-effectiveness of caching for these jobs. In this scenario, it uses a multi-job

handling algorithm to adjust the importance values of samples in H-list to reduce the total training time of all the jobs.

H-cache. It stores H-samples recorded in H-list. Its capacity determines how many data items can be stored in H-cache. When its capacity is not large enough to cache all H-samples, an importance-informed cache replacement algorithm is applied to manage it. The general idea is that the data items of higher importance value have a lower chance of being evicted from the cache. And the importance value is provided by H-list. If the importance values of H-samples are changed leading to a lower hit ratio of H-cache, it needs to refill the cache with new H-samples. When importance sampling is used for training, the importance-informed cache replacement achieves a higher cache hit ratio than the commonly used LRU-based cache replacement algorithm and its variants.

The size of H-cache is determined based on the following equation: $Size_{hcache} = Size_{cache} * \frac{Frequence_{HI}}{Frequence_{LI}+Frequence_{HI}}$. $Frequence_{HI}$ and $Frequence_{LI}$ are the frequency of accesses to H-samples and L-samples respectively. $Size_{cache}$ is the cache size. A higher $\frac{Frequence_{HI}}{Frequence_{LI}}$ automatically increases the cache space allocated for H-samples and reduces the space for L-samples. The iCACHE manager tracks the number of accesses to H-samples and L-samples. The minimum size of L-cache is equal to the number of data items in one package.

L-cache. The purpose of L-cache is to further reduce the number of small random I/Os for accessing L-samples. It is designed to cache only the L-samples, which are not in the H-list. Another benefit of L-cache is to maintain a high model accuracy. Basically, we manage L-cache with *substitutability*, a unique characteristic of the DNN I/O process [27]; it means when a read request is missed in the cache, it can be served by another randomly picked sample in the cache. While the missed samples from L-cache can be substituted with the one in H-cache, serving them with another one in L-cache can keep a high degree of sample diversity and yields better training accuracy (See Section V-E). The L-samples are packaged in advance by an asynchronous thread. In addition, because the importance values of data items are constantly changing across epochs, it also needs to re-packing the L-samples accordingly. The size of L-cache is equal to $Size_{cache} - Size_{hcache}$.

B. Importance-Informed Cache Algorithm

We use a key-value store to manage data items in H-cache. The key denotes sample ID and the value stores the data item of a sample. iCACHE also manages a small-top-heap (H-heap) for cache management. The heap objects are also key-value pairs, whose key is the importance value of a data item and value is a reference to the item in the key-value store. The objects in H-heap are sorted based on their importance values. The object at the top of the heap is called *top-node*. The size of a heap object is 16 B. The space usage of H-heap is correlated to the number of H-samples and is generally less than 0.5% of H-cache capacity.

When the requested data items are H-samples but do not exist in H-cache, the server needs to read them from storage systems and return them to clients. It also needs to decide whether to cache the sample when H-cache is full. One challenge is that LRU-based cache replacement algorithms do not work effectively in training with importance sampling because they do not consider sample importance. Our observation shows that H-samples are accessed more frequently than L-samples in each epoch and across multiple epochs. Therefore, we need a new importance-informed cache replacement to improve the cache hit ratio in training.

Algorithm 1 Importance-Informed Caching Algorithm.

Require: bs: batch size;
$batch\_id$: sequence of training sample's ID for one batch produced by IIS;
$H-list$: the current H-list in server, which is pulled from the client;

```
batch_data ← []
for i ← 0, 1, ..., (bs − 1) do
    id ← batch_id[i]
    if id in H − list then                    ▷ The sample is an H-sample.
        if id in H_cache then
            data ← read_from_H_cache(id)
        else
            data ← read_from_storage(id)
            if H_cache is not full then
                insert_sample_into_H_cache(id, data)
            else
                iv_cur ← get_importance(id)
                iv_min ← min_imp_in_H_heap()
                if iv_cur > iv_min then
                    delete_and_insert_H_cache(id, data)
                end if
            end if
        end if
    else                                      ▷ The sample is an L-sample.
        if id in L_cache then
            data ← read_from_L_cache(id)
        else                                  ▷ Substitution.
            data ← randomly_fetch_from_L_cache()
        end if
    end if
    batch_data.append(data)
end for
    return batch_data
```

Specifically, when H-cache is not full, the H-sample read from storage systems will be inserted into H-cache directly. iCACHE then creates a heap object corresponding to the H-sample and inserts it into the heap. When it is full, the importance value of an incoming H-sample is compared to that of top-node. The top-node will be evicted if its importance value is smaller than that of the incoming sample. Otherwise, the incoming sample will not be admitted. If the top-node is evicted, iCACHE will create a new heap object corresponding to the incoming H-sample and insert it into the heap. A high-level algorithm for the importance-sampling cache management is shown in Algorithm 1.

Importance-informed cache replacement algorithm performs better than traditional LRU-like algorithms exploiting temporal and spatial locality. Let's take Figure 5 as an example. We assume that the capacity of H-cache is three and three data items (#1, #2, #3) have been cached consecutively. When item

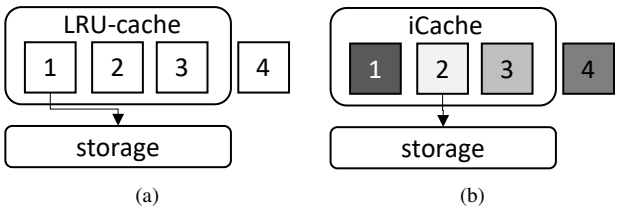


Fig. 5. The comparison of LRU and iCACHE. The squares in darker color denote samples with higher importance values.

#4 is accessed, the LRU-like replacement algorithm will evict item #1 because it is least recently used (shown in Figure 5(a)). If we further assume item #2 has the least importance value thus is the top-node in the heap, we will have a higher probability of accessing #1 than accessing #2 in future references. Hence, a higher cache hit ratio will be achieved by evicting sample #2 (shown in Figure 5(b)). We will experimentally demonstrate the effectiveness of importance-informed cache over LRU-based cache in Section V-C.

The second challenge in the design of iCACHE is the importance values of data items change as the model is trained, leading to variation of a cache hit ratio. To solve this issue, iCACHE periodically updates H-list by pulling it from cache clients. Because of the overhead of building a small-top-heap, we do not update H-heap in place. Instead, iCACHE manages a shadow heap, which has the same structure as H-heap. After H-list is updated, the current H-heap becomes read-only and we continue using it for eviction purposes. But new heap objects are inserted into the shadow heap with their updated importance value. After the shadow heap is rebuilt completely against the updated H-list, it can be directly used as a new H-heap. Then the original H-heap is released. In this way, the update of importance values in the heap can be performed asynchronously and does not affect the critical I/O path of the training process.

C. Dynamic Packaging

Because iCACHE only stores H-samples in H-cache, L-samples that are not in H-list may still incur small random I/Os. We design a new approach, named dynamic packaging, to reduce data stall time caused by accessing L-samples. The idea is to maintain a small L-cache for storing L-samples. The L-samples are loaded in the memory in the unit of a package to improve I/O efficiency. The package size is at least 1 MB exploiting the spatial locality of storage systems. When an L-sample is requested, iCACHE returns the data item from L-cache if it is a hit. Otherwise, instead of reading the requested L-sample from the storage system, we apply substitutability and return a cached L-sample that has not been accessed at the current epoch. The IDs of L-samples that are missed in the cache will be recorded and later loaded from storage systems by the loading thread. Our results show that applying substitutability on L-samples has a very minor impact on the model accuracy while significantly reducing data stall time. Because we replace any missed L-samples with other L-samples in L-cache, we can achieve a hit ratio of 100% in L-cache. Consequently, L-heap is not needed for L-samples.

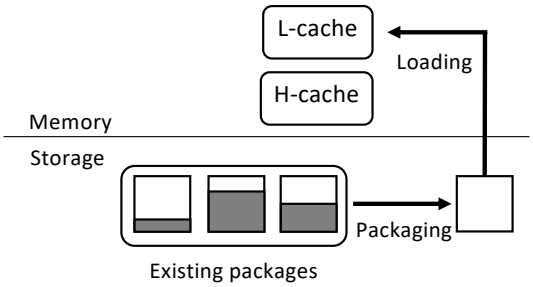


Fig. 6. Illustration of packing thread and loading thread. The white area in the package represents samples with low importance values and the dark area denotes samples with high importance values.

Specifically, iCACHE uses two concurrent threads (i.e., packaging thread and loading thread) working together to achieve dynamic packaging, as shown in Figure 6. At the beginning of epoch 1, there are no packages in the storage system. Once H-list is generated, the packaging thread randomly selects L-samples. It then packs them into large file packages and stores them in the storage system. Then the loading thread chooses one package and caches all data items in the package in L-cache. Next, when a requested sample is not in H-list and missed in L-cache, iCACHE directly randomly chooses a cached L-sample which has not been accessed to replace the requested one. When all data items in L-cache have been accessed once, new packages will be read into L-cache by the loading thread.

After H-list is updated, the ratio of L-samples in the existing packages is changed in the following training epochs. We need to periodically repack the samples to make sure that a package consists of a large number of L-samples to fill L-cache quickly in fewer I/Os. To achieve this goal, the samples that are previously missed in L-cache will be re-packed in the packages to increase sample diversity. Then the rest of space will be filled with L-samples that are randomly selected from the existing packages. At the same time, the loading thread will load the reorganized packages into memory. Then it stores L-samples in L-cache.

Both the loading thread and packaging thread run asynchronously. The package reorganization introduces three benefits. First, L-samples can be read in the form of large packages, which alleviates random small I/Os. Second, because of the dynamic repackaging, we can guarantee that a large number of L-samples are stored into L-cache for every I/O, thus improving the effective storage bandwidth. Third, compared to the existing fixed packaging strategy [2], package reorganization may improve model accuracy because it can increase the randomness of samples being trained.

D. Multi-Job Handling

When multiple jobs use the same dataset for DNN model training, the same training sample may be repeatedly accessed by all the jobs. However, iCACHE may receive different importance values of the same data item for two reasons. First, different DNN model architectures lead to different fit capabilities for the same data item. Second, the importance values of samples generally tend to decrease as the training proceeds

because the loss decreases as the training converges [11]. Some jobs that require less computation time may proceed to the next training epochs earlier than other jobs. These jobs will generate smaller importance values corresponding to the same samples.

To address this issue, we design a module in iCACHE for multi-job handling. It periodically evaluates the cost-effectiveness of caching for each job. We define cache-eligible jobs as the jobs that are deemed to benefit from cache. Then it computes an adjusted importance value for each data item given all the importance values collected from the cache-eligible jobs. Finally, the adjusted importance value will be used by iCACHE for cache management (e.g., heap rebuilding and cache eviction).

Cache benefit estimation. At the beginning of each epoch, iCACHE uses 40 mini-batches to estimate the caching benefit ($Ratio_{benefit}$) which is quantified as a ratio of execution time without cache $T_{cacheless}$ and execution time with cache T_{cache}. $T_{cacheless}$ is measured when all I/Os are served on storage systems without cache for the first 20 mini-batches. T_{cache} is measured when iCACHE is enabled for the job for the second 20 mini-batches. For a particular job, if its $Ratio_{benefit}$ is higher than a threshold (1.5 in our current design), it is deemed as cost-effective and becomes cache-eligible. Then iCACHE will use its H-list in computing the adjusted importance value.

Adjusted importance value computation. After receiving H-lists from all the concurrent jobs accessing the same dataset, we compute the relative importance value of every data item in two steps. (1) We compute the relative importance value of data item i for each training job using the percentile position of IV_i in the whole training set and denote it with RIV_i. (2) For data item i, we compute its $AIV_i = \sum_{j=0}^{N-1} Ratio_{benefit}^{j} * RIV_i^j$ where AIV_i is the aggregated importance value of data item i, N is the number of concurrent jobs that access the dataset and are cache-eligible, $Ratio_{benefit}^{j}$ is the caching benefit of job j, and RIV_i^j is the relative importance value of data item i from job j.

E. Distributed iCACHE

A distributed deep learning training application usually uses data parallelism to achieve training across multiple nodes. To support this scenario, we extend iCACHE to a distributed cache. Each node is equipped with a local cache client, a cache server, and a cache manager whose functionalities are introduced in III-A. To cache as much data as possible, the data item stored by each node is not duplicated. To determine on which node a specific data item is cached, a distributed key-value store is shared among all training nodes, which records key-value pairs formed by the data item ID and the node ID where it is located.

The ideas of I/O-oriented importance sampling, importance-informed cache, dynamic packaging, and multi-job handling in a single node can be naturally applied to the distributed cache system. Besides, the distributed cache needs to consider remote cache in the data flow of the distributed cache management. Specifically, when a cache client requests a data item, the cache manager will first check whether the data exists in the local cache. If so, it returns the data from the local node; otherwise, the cache manager will judge whether the data exists in the cache space of other nodes by looking up the distributed key-value store. If it exists, the data item will be read from other nodes; otherwise, the request will be forwarded to the shared underlying storage system.

IV. IMPLEMENTATION

We implement the client of iCACHE in Python based on PyTorch 1.8.0 [40]. We provide a new $iCacheImageFolder$ interface in the original $torch.utils.Dataset$ class, which uses the gRPC [46] framework to communicate with the iCACHE server. The client gets samples from the server through the rpc_loader interface and sends the H-list of samples to the server through the $update\_ivpersample$ interface. For the server, we implement it in Go language. We use the key-value structure to organize the samples in H-cache and L-cache. In addition to providing the usual functions of lookup/access/insert, the server also provides an interface to receive importance values and modules to handle dynamic packaging and multi-job coordination in cache management. In all, the client is implemented with around 2000 LOC and the server is with around 3500 LOC. iCACHE is easy to deploy. Users only need to replace the original $Dataset.ImageFolder$ interface with $Dataset.iCacheImageFolder$ after starting the iCACHE server with a Go command.

V. EVALUATION

A. Experimental Setup

System configurations. We conduct the experiments on a training server with 2× AMD EPYC 7742 CPUs (64 cores), 512 GB DRAM, 10Gbps Ethernet, 8× NVIDIA A100 GPUs, one Intel SSD of 1 TB. The operating system is 64-bit Ubuntu 18.04.5. We train the DNN models using PyTorch 1.8.0 on the server and place the training datasets in an OrangeFS parallel file system [1] in the same data center.

Workloads and datasets. We use a small dataset CIFAR10 [26] and a large dataset ImageNet [10]. With CIFAR10, we train ShuffleNet [50], ResNet18 [13], MobileNet [14], and ResNet50 [13]. With ImageNet, we train VGG11 [22], MnasNet [43], SqueezeNet [16], and DenseNet121 [15]. These eight models are widely used in the prior work [27], [29], [36]. We do not consider natural language processing models (e.g., BERT) because they are typically computation-bound [36].

Compared systems. We compare iCACHE with Default, Base, Quiver [27], CoorDL [36], and iLFU. Default represents the existing PyTorch framework with a user-level LRU cache. Base is the version with the user-level LRU cache and the computing-oriented importance sampling (CIS) to only reduce computing. Both Base and iCACHE use the loss-based algorithm [18] to determine the sample importance. Quiver performs similarly to Default but sample substitutability is

TABLE I
MODEL ACCURACY ON CIFAR10.

Models	Top-1 Acc.(%)		Top-5 Acc.(%)	
	Default	iCACHE	Default	iCACHE
ShuffleNet	87.76	86.96	99.59	99.57
ResNet18	92.70	92.14	99.81	99.80
MobileNet	92.37	92.01	99.87	99.77
ResNet50	89.91	89.36	99.68	99.69

TABLE II
MODEL ACCURACY ON IMAGENET.

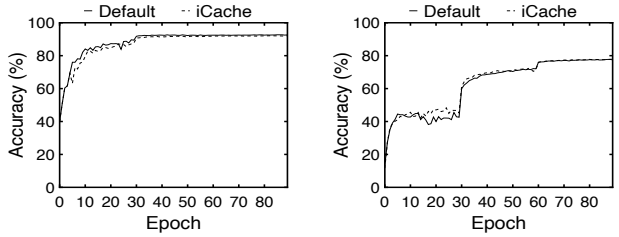
Models	Top-1 Acc.(%)		Top-5 Acc.(%)	
	Default	iCACHE	Default	iCACHE
VGG11	67.06	65.67	87.46	86.13
MnasNet	58.59	57.25	81.78	80.16
SqueezeNet	54.69	53.83	77.72	77.91
DenseNet121	75.35	74.79	92.57	92.39

used to enhance cache management. It also proposes coordinated eviction and co-operative cache handling for multi-job training. CoorDL is based on Default but it does not evict the already cached data in each epoch. Besides, it allows pre-processed data in memory to be shared among multiple jobs. To demonstrate the efficiency of iCACHE over traditional caching methods, we also implement iLFU, which applies the I/O-oriented importance sampling (IIS) with the least frequently used (LFU) cache management. In contrast, iCACHE applies IIS and manages the cache with our designs. Besides, we add the Oracle comparison which means all accesses to the cache are hit to show the lower bound of training time. As Quiver is not open-source, we re-implement it as faithfully as possible according to the descriptions in the paper. For CoorDL, we evaluate it by referring to its open-source implementation [35]. We enable prefetching in PyTorch in all experiments.

By default, the cache size is 20% of the dataset, the initial ratio of $Size_{hcache}$ to $Size_{lcache}$ is 9:1, the number of workers to fetch data is 6, the batch size is 256, and the training datasets are striped over four servers with a stripe size of 64 KB in OrangeFS.

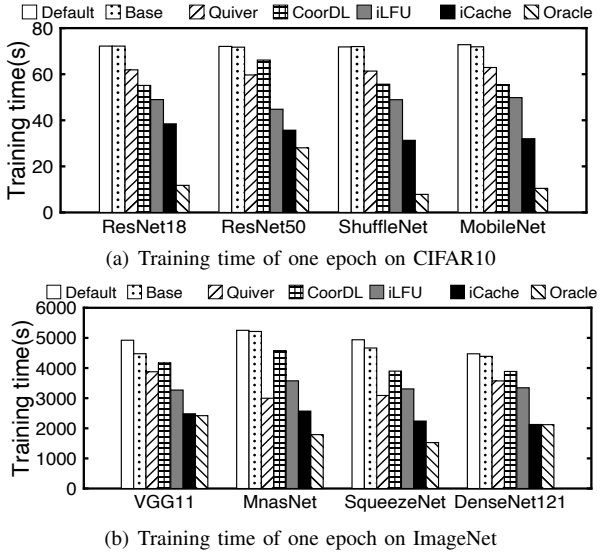
B. Accuracy Results

We first present the accuracy comparison for the four models on CIFAR10 with different cache schemes. Table I shows iCACHE achieves the comparable Top-1 and Top-5 accuracy compared to Default for all models. More specifically, iCACHE has 0.80%, 0.56%, 0.36%, and 0.55% accuracy losses on Top-1 accuracy and 0.02%, 0.01%, 0.10%, and -0.01% accuracy losses on Top-5 accuracy for ShuffleNet, ResNet18, MobileNet, and ResNet50, respectively. The accuracy loss is constrained within 1%. Table II shows the model accuracy on ImageNet. The accuracy loss of iCACHE yields satisfactory accuracy with less than 2% losses compared to Default.



(a) ResNet18 on CIFAR10 (b) SqueezeNet on ImageNet

Fig. 7. The Top-5 accuracy comparison.



(a) Training time of one epoch on CIFAR10

(b) Training time of one epoch on ImageNet

Fig. 8. The training time per epoch.

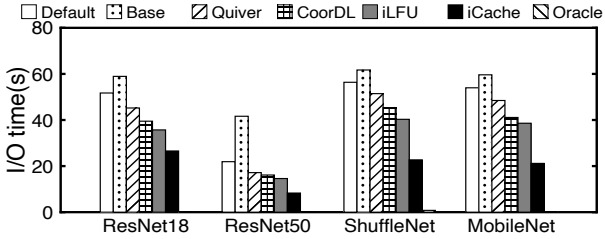


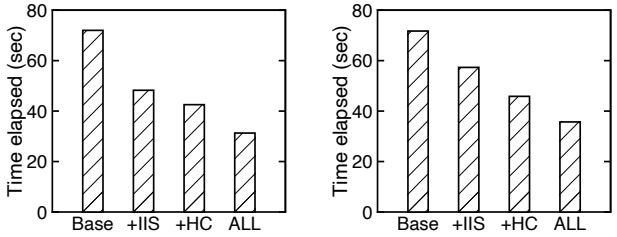
Fig. 9. I/O time of training one epoch on CIFAR10.

Figure 7(a) and Figure 7(b) plot the Top-5 accuracy convergence curves for ResNet18 on CIFAR10 and SqueezeNet on ImageNet in 90 epochs, respectively. We can see that the curves of iCACHE are closely matched with the curves of Default, which provides the highest accuracy.

C. Performance Results

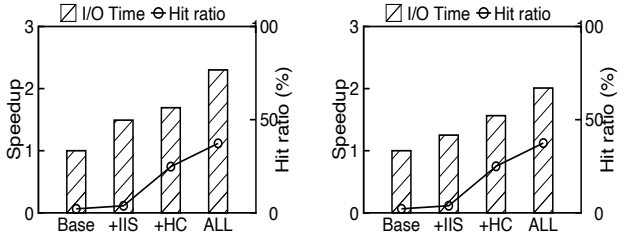
Figure 8 shows the average training time per epoch for the DNN models. The average training time is defined as the total training time divided by the number of epochs. We have three observations.

First, iCACHE outperforms all other five cache systems. On CIFAR10, it achieves maximum speedups of 2.3×, 2.3×, 2.0×, 1.9×, 1.6× over Default, Base, Quiver, CoorDL, and iLFU for the four models, respectively. On ImageNet, the maximum speedups are 2.2×, 2.1×, 1.7×, 1.8×, and 1.5× for the other four models. We also observe that on VGG11 and DenseNet121, iCACHE performs almost the same as Oracle. This shows iCACHE can efficiently alleviate the I/O bottleneck of these two models.



(a) ShuffleNet on CIFAR10. (b) ResNet50 on CIFAR10.

Fig. 10. The performance impact on total training time with IIS, H-Cache, and L-cache in iCACHE.



(a) ShuffleNet on CIFAR10. (b) ResNet50 on CIFAR10.

Fig. 11. The impact on I/O time and hit ratio with IIS, H-Cache, and L-cache in iCACHE.

Second, different DNN models yield varied performance improvements. For example, iCACHE achieves the highest improvement (i.e., 2.3×) for ShuffleNet on CIFAR10, but smaller speedups for other three models. This is because ShuffleNet requires less GPU computation than other models, making the training more I/O bound.

Third, Base has the minimum performance benefit, compared to Quiver, CoorDL, iLFU, and iCACHE. This is because Base can only reduce computation time with CIS while the other four can further reduce I/O time in the I/O bound model training.

Figure 9 presents the I/O time per epoch in DNN training. Because of the space limitation, we only show the results for CIFAR10. Compared to Default, iCACHE reduces the I/O time by 2.4× on average for the four models, while Quiver, CoorDL, and iLFU achieve an average speedup of 1.2×, 1.3×, and 1.4× respectively. The benefit of iCACHE comes from the reduced data fetches using IIS and the improved cache hit ratio using the importance-aware cache management policies. Although iLFU also reduces data fetches by IIS, the LFU policy is reactive to the changes of sample importance, resulting in a lower cache hit ratio. This explains why iCACHE has the best performance. Note that Base has a 1.3× longer average I/O time than Default. This is because Base only reduces the computing time using CIS and does not change the data fetching time, resulting in less I/O time hidden by the computing time in the training pipeline.

D. Impact of Individual Techniques

Figure 10 shows the impact of each optimization in iCACHE on the total training time. *Base* is the system with the computing-oriented importance sampling (CIS) and an LRU cache (CIS+LRU). *+IIS* denotes the version where I/O-oriented importance sampling (IIS) is used to reduce the number of fetches. *+HC* denotes the variant where H-cache is managed according to sample importance. *All* denotes the version with all optimizations including L-cache is enabled. In the experiments, we train ShuffleNet and ResNet50 with CIFAR10.

Figure 10(a) shows *+IIS* achieves a speedup of 1.4× over *Base* for ShuffleNet. The primary reason is that *Base* can only reduce the computation while *+IIS* can further reduce the number of I/Os by up to 31.4%. When H-cache is enabled, the importance-sampling-informed algorithm caches more H-samples and significantly increases the hit ratio in iCACHE. Thus, *+HC* achieves a speedup of 1.7× compared to *Base*. When L-cache is further enabled, *All* achieves a speedup of 2.3×. This is because L-samples that are missed in memory are substituted by other L-samples in L-cache, further reducing the data loading time.

Figure 11 demonstrates the I/O time and cache hit ratio with individual techniques. Figure 11(a) shows that the cache hit ratio has increased significantly for ShuffleNet on CIFAR10. When enabling the HC technique, the hit ratio is increased from 2% to 25%. This is because *+HC* increases temporal locality, thus H-samples are more likely to be kept in H-cache without cache thrashing. After L-cache is enabled, the hit ratio is further increased to 37%. This is because L-samples can be substituted with others in L-cache.

We have similar observations on the trend of speedup and cache hit ratio for ResNet50 on CIFAR10, as shown in Figure 10(b) and Figure 11(b).

TABLE III
MODEL ACCURACY ON CIFAR10.

Models	Top-1 Acc.(%)			Top-5 Acc.(%)		
	Def	ST_{HC}	ST_{LC}	Def	ST_{HC}	ST_{LC}
ResNet18	92.70	91.89	92.14	99.81	99.77	99.80
ShuffleNet	87.76	86.73	86.96	99.59	99.52	99.57

E. Impact of Sample Substitution on Model Accuracy

To accelerate model training without significant accuracy degradation, iCACHE does not substitute H-samples with other samples when they are missed in the cache. However, when L-samples are missed, we can substitute them with samples in either H-cache or L-cache. Since either case shows the same I/O performance, we study its impact on model accuracy.

Table III shows the model accuracy with different sample substitution policies. Def denotes the policy without sample substitution. ST_{HC} and ST_{LC} represent the policy to substitute the missed sample with H-sample and L-sample respectively. For ResNet18, the Top-1 model accuracy drops 0.81% and 0.56% with ST_{HC} and ST_{LC} compared to Def. For ShuffleNet, ST_{HC} and ST_{LC} yield a 1.03% and 0.80% Top-1 accuracy drop. These results show ST_{LC} has less impact on model accuracy. Similar trends can be observed in Top-5

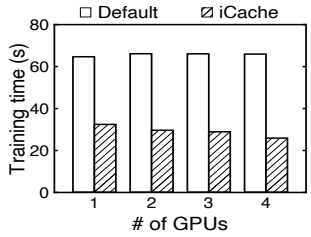


Fig. 12. Performance of iCACHE on multi-GPUs.

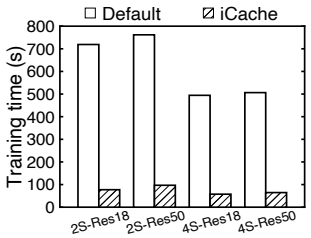
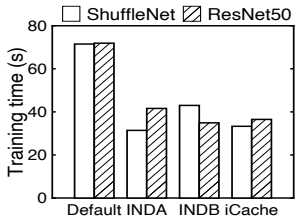
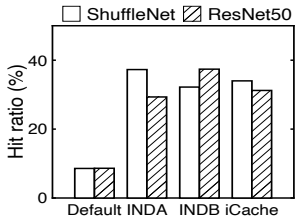


Fig. 13. Distributed training on CIFAR10. The label "nS" means the system with "n" servers.



(a) Time.



(b) Hit ratio.

Fig. 14. The average training time per epoch and cache hit ratio with different caching schemes for multi-job execution.

accuracy for other models on CIFAR10 and ImageNet. Thus, iCACHE takes this substituting method.

F. Single-Job Multi-GPU Training

Figure 12 shows the per-epoch training time of ResNet50 on multi-GPUs with CIFAR10. We can find iCACHE always takes less training time than $Default$ for all GPU configurations. Compared to $Default$, iCACHE achieves an average speedup of 2.3× with different numbers of GPUs. This shows iCACHE is effective in multi-GPU training. Another observation is that the total training time of $Default$ remains similar as the number of GPUs increases. This is because increasing GPUs reduces a small amount of computing time but also incurs the communication overhead among multiple GPUs. In contrast, iCACHE has a slight performance improvement because it can reduce a lot of I/O time with the IIS and importance-aware cache management techniques, making the training process less I/O bound.

G. Multi-Server Distributed Training

Figure 13 shows the system performance of iCACHE in a distributed cloud platform with two and four nodes. Each server is equipped with one GPU and a cache space of 20% of the whole training data. As we do not have the permission to install the kernel module of OrangeFS on the cloud platform, we store the training data in an NFS server, similar to the approach in other cloud deep learning systems [17], [38], [42]. The maximum read bandwidth of the NFS is about 10Gb/s. Although our scale is small, we argue that it is sufficient to demonstrate the efficiency of our system. Due to space limitations, we only show the results on CIFAR10 for ResNet18 and ResNet50. The results of other models lead to similar observations as described below.

First, iCACHE performs better in distributed training than Default. Specifically, iCACHE speeds up at least 8.6× and 7.6× under 2-server and 4-server configurations. The main reasons are similar to that of the single-server training scenario described in Section V-C. Besides, we observe that the 4-server training time is lower than 2-server time by around 1.5x.

Second, the speedup of iCACHE with the 4-server configuration is less than that with 2-server setting. For instance, the speed up is reduced from 9.3× to 8.5× for ResNet18. This is because a larger joint cache space results in a smaller improvement of cache hit ratio. Specifically, the cache hit ratio is increased by 42% and 23% on 2-server and 4-server, respectively.

H. Multi-Job Training

To demonstrate the effectiveness of the multi-job handling module, we run ShuffleNet and ResNet50 simultaneously on the same CIFAR10 dataset. These two jobs share the cache space. We evaluate the performance of each job with four caching schemes. They are $Default$ (using LRU to manage cache space), $INDA$ (using the importance values from an individual model, i.e., ShuffleNet, to make caching decisions), $INDB$ (using the importance values from ResNet50), and iCACHE with the policy discussed in Section III-D.

Figure 14(a) shows the average training time per epoch of each job. We have three observations. First, $INDA$, $INDB$, and iCACHE perform better than $Default$. This is because the former three schemes reduce the number of samples fetched from storage and improve the cache hit rate. Second, for $INDA$ and $INDB$, they only perform well for the individual model but are suboptimal for the other model. For example, compared to $INDB$, $INDA$ speeds up the ShuffleNet training by $1.4\times$ but it slowdowns ResNet50 by $1.2\times$. This is because the cache gives higher priority to data items from one job, causing a higher cache miss ratio for the other job. Third, iCACHE achieves the minimum system completion time for both jobs among the four schemes. Specifically, it achieves $1.1\times$ and $1.2\times$ speed up compared to $INDA$ and $INDB$, respectively. This result shows the efficiency of the multi-job handling module in iCACHE.

Figure 14(b) presents the cache hit ratio for each job. We observe that ShuffleNet has a higher hit ratio than ResNet50 when using iCACHE. This is because iCACHE automatically perceives that ShuffleNet has a heavier I/O bottleneck than ResNet50, and the benefit of caching is higher.

I. Parameter Sensitivity Analysis

Number of prefetching workers. PyTorch employs multiple workers to prefetch training data from storage systems. Figure 15 shows the training time per epoch with various number of workers. We train ResNet18 on CIFAR10. As shown, iCACHE achieves a speedup over Default from 3.9× to 1.2× while the number of workers is increased from 2 to 16. This is because the proportion of data stall time decreases from 96.7% to 28.9% when the number of workers increases. Thus, the I/O benefits brought by iCACHE diminishes. However,

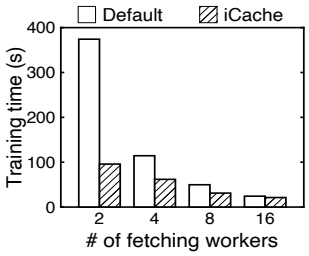


Fig. 15. Performance impact of number of workers.

Cache	Speedup	Hit ratio	
		Ours	Default
20%	1.7 ×	0.37	0.02
40%	1.9 ×	0.55	0.09
60%	2.1×	0.69	0.23
80%	2.4×	0.83	0.48

Fig. 16. Performance impact of cache size.

since NVIDIA's AI-optimized servers (i.e., DGX-2) or general commercial cloud servers typically provide users with 3-4 CPU cores (6-8 vCPUs) per GPU [36], the number of workers set by users are usually limited to eight. Therefore, the prefetching effect is limited and iCache is still useful.

Cache size. Figure 16 shows the training performance with various cache sizes for ResNet18 on CIFAR10. First, we can observe that with iCACHE the minimum speedup of training time is 1.7× as the cache size increases from 20% to 80%. This demonstrates that iCACHE is effective for reducing data stall time when the cache sizes are varied. Second, the cache hit ratio increases as the cache size increases for both iCACHE and Default. However, even when the cache size is 80% of the dataset size, iCACHE still achieves a 1.7× higher cache hit ratio than Default. This explains why iCACHE has a superior performance.

VI. DISCUSSION

PM-based cache. iCACHE currently builds the cache using DRAM, but it is not limited to that; emerging large-capacity persistent memory (PM) is another option. However, PM requires specific hardware (e.g., CPUs) [8], [45] and it has relatively lower performance than DRAM [30], thus DRAM is still the dominant memory. We will explore the PM-based cache in future work.

Other importance sampling methods. Although we choose the loss-based algorithm [18] to evaluate the importance of samples, other algorithms (e.g. using a lightweight model for evaluation [49]) can also be modified and integrated into iCACHE. Specifically, we can still decide which samples will be trained before the start of each epoch training based on the historical importance value, reducing the data to be fetched from the storage layer. We can also implement the aforementioned techniques to increase the cache hit rate.

Local storage cache. Although iCACHE only uses DRAM as cache space and does not use local storage, this does not affect the validity of the core idea of iCACHE. In addition, AI training data in industry often cannot be fully cached in local storage [9]. For example, the production training dataset of recommendation system DNN models in Meta can reach 29.18PB [52], but the local storage capacity in each node is typically only a few TB at most, necessitating the need to retrieve a large amount of data from remote storage servers or parallel file systems.

VII. RELATED WORK

A. Cache Management for DNN Training

Existing cache optimizations for DNN include expanding the cache capacity, improving the cache hit ratio considering the access pattern of DNN training, and sharing the data in the cache between training jobs.

Considering larger cache can hold more training samples from a back-end parallel file system, DeepIO [53] and CoorDL [36] use RDMA or TCP to connect DRAM of multiple nodes to form a shared memory to expand the cache capacity. Fanstore [51] forms a global cache layer on burst buffers across nodes via MPI. iCACHE is a specially designed single-node cache system but can be easily extended to a multi-node environment.

To improve the cache hit ratio, CoorDL [36] and MONARCH [9] do not evict the already cached samples because all samples are accessed in the training process in each epoch. This method has a limited cache hit ratio, which is determined by the ratio of cache capacity and dataset size. Quiver [27] proposes to replace the missed samples with samples in the cache applying the substitutability of data items in training. In contrast, iCACHE only substitutes unimportant samples (i.e., L-samples) missed in the cache because replacing important samples changes the distribution of H-samples decided by importance sampling algorithms, which may impact the final model accuracy. The latter is a key to accelerating training convergence and maintaining training accuracy.

When multiple jobs are trained on the same dataset, a unified cache is designed to share data items between the jobs in OneAccess [21] and CoorDL [36]. iCACHE only shares samples that all jobs consider important. Not only does our approach reduce data redundancy in the cache but also accelerates the convergence of I/O-bound jobs with negligible accuracy loss of DNN models.

B. Storage Optimizations for DNN Training

To solve the performance issues caused by randomly reading small samples from storage systems, much work uses static data packaging, such as TFRecord in TensorFlow [2], Webdataset in AIStore [3], chunk data in DIESEL [44], Quiver [27], and DLFS [54]. Different from the existing approaches which pre-packs all data, we use dynamic packaging in iCACHE. It selects unimportant samples to pack at runtime. Thus, samples in packages are not static, thus improving the randomness of data access in training.

Google proposes a technique called data echoing [6] to eliminate the impact of data stalls caused by inefficient access from storage. The main idea is reusing fetched data while waiting for the next training batch to be loaded into memory. MET [47] employs a similar concept and presents techniques to automatically tune reusing factors. Meta proposes feature flattening, coalesced reads, feature reordering, and other techniques to improve the data storage and ingestion pipeline of recommendation model training [52]. ReFlex [25] allows

clients to access remote flash with high performance as local flash, which can be used in deep learning to speed up remote training data accesses. These techniques are orthogonal to our work and can be combined to further reduce stress on storage systems.

C. Other Works in Context

DLFS [54] is a user-level, read-optimized file system for deep learning that accesses local or remote targets in an OS-bypass manner. It also accelerates metadata retrieval by making each node maintain global metadata. Yang et al. [48] proposed a locality-aware data loader which re-distributes data among nodes in distributed data-parallel training to reduce data communication when fetching data from remote nodes. Macedo et al. [33] decouples storage optimizations from deep learning frameworks to enhance the applicability and portability of the optimizations, getting coordinated and holistic control of global resources. These optimization methods are also orthogonal to our work.

VIII. Conclusion

In this paper, we describe the design and implementation of iCache, a novel cache system to reduce data stall time for I/O-bound DNN training jobs. We are the first to propose I/O-oriented importance sampling and apply it to cache systems. iCache consists of both H-cache and L-cache, which store samples with high importance and low importance respectively. We use importance values of samples to manage data items in H-cache to improve the cache hit ratio. When there is a cache miss in L-cache, we use sample substitutability and dynamic packaging to reduce the number of random I/Os. Finally, it provides multi-job coordination to avoid cache thrashing considering the cost-effectiveness of caching of jobs. Our experimental results show that iCache has a negligible impact on training accuracy and speeds up the DNN training time by up to 2.0× compared to the state-of-the-art caching systems. As deep learning applications become more and more popular, we hope iCache will inspire the next generation of memory and storage systems designed for artificial intelligence.

Acknowledgments

We sincerely thank the anonymous reviewers for their constructive suggestions. This work was supported in part by the National Key Research and Development Program of China No.2021ZD0110700, the National Science Foundation of China No. 62172361, the Program of Zhejiang Province Science and Technology No. 2022C01044, the Zhejiang Lab Research Project No. 2020KC0AC01, and the US National Science Foundation under CNS 1906541.

References

[1] "Orange File System," http://www.orangefs.org/, 2021.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A System for Large-Scale Machine Learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 265–283.

[3] A. Aizman, G. Maltby, and T. Breuel, "High Performance I/O for Large Scale Deep Learning," in *Proceedings of the International Conference on Big Data*, 2019, pp. 5965–5967.

[4] P. Chen, S. He, X. Zhang, S. Chen, P. Hong, Y. Yin, and X.-H. Sun, "Accelerating Tensor Swapping in GPUs With Self-Tuning Compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4484–4498, 2022.

[5] P. Chen, S. He, X. Zhang, S. Chen, P. Hong, Y. Yin, X.-H. Sun, and G. Chen, "CSWAP: A Self-Tuning Compression Framework for Accelerating Tensor Swapping in GPUs," in *IEEE International Conference on Cluster Computing*. IEEE, 2021, pp. 271–282.

[6] D. Choi, A. Passos, C. J. Shallue, and G. E. Dahl, "Faster Neural Network Training With Data Echoing," *arXiv preprint arXiv:1907.05550*, 2019.

[7] J. Chung, K. Lee, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Ubershuffle: Communication-Efficient Data Shuffling for SGD via Coding Theory," *Advances in Neural Information Processing Systems*, 2017.

[8] Z. Dang, S. He, P. Hong, Z. Li, X. Zhang, X.-H. Sun, and G. Chen, "NVAlloc: Rethinking Heap Metadata Management in Persistent Memory Allocators," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 115–127.

[9] M. Dantas, D. Leitao, C. Correia, R. Macedo, W. Xu, and J. Paulo, "MONARCH: Hierarchical Storage Management for Deep Learning Frameworks," in *Proceedings of the 2021 IEEE International Conference on Cluster Computing*. IEEE, 2021, pp. 657–663.

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-Scale Hierarchical Image Database," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[11] I. Goodfellow, Y. Bengio, and A. Courville, "Deep learning: Adaptive Computation and Machine Learning Series," *Cambridge Massachusetts*, 2017.

[12] Google, "Open images dataset," https://github.com/cvdfoundation/open-images-dataset, 2018.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[14] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.

[15] F. Iandola, M. Moskewicz, S. Karayev, R. Girshick, T. Darrell, and K. Keutzer, "Densenet: Implementing Efficient Convnet Descriptor Pyramids," *arXiv preprint arXiv:1404.1869*, 2014.

[16] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-Level Accuracy with 50x Fewer Parameters and¡ 0.5 MB Model Size," *arXiv preprint arXiv:1602.07360*, 2016.

[17] K. R. Jayaram, V. Muthusamy, P. Dube, V. Ishakian, C. Wang, B. Herta, S. Boag, D. Arroyo, A. Tantawi, A. Verma, F. Pollok, and R. Khalaf, "FfDL: A Flexible Multi-Tenant Deep Learning Platform," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 82–95.

[18] A. H. Jiang, D. L.-K. Wong, G. Zhou, D. G. Andersen, J. Dean, G. R. Ganger, G. Joshi, M. Kaminksy, M. Kozuch, Z. C. Lipton, and P. Pillai, "Accelerating Deep Learning by Focusing on The Biggest Losers," *arXiv preprint arXiv:1910.00762*, 2019.

[19] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," *SIGMETRICS Perform. Eval. Rev.*, 2002.

[20] T. B. Johnson and C. Guestrin, "Training Deep Models Faster with Robust, Approximate Importance Sampling," *Advances in Neural Information Processing Systems*, 2018.

[21] A. Kakaraparthy, A. Venkatesh, A. Phanishayee, and S. Venkataraman, "The Case for Unifying Data Loading in Machine Learning Clusters," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing*, 2019, pp. 283–296.

[22] Karen Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *arxiv: 1409.1556*, 2014.

[23] A. Katharopoulos and F. Fleuret, "Biased Importance Sampling for Deep Neural Network Training," *arXiv preprint arXiv:1706.00043*, 2017.

[24] A. Katharopoulos and F. Fleuret, "Not all Samples are Created Equal: Deep Learning with Importance Sampling," in *Proceedings of the International Conference on Machine Learning*, 2018, pp. 2525–2534.

[25] A. Klimovic, H. Litz, and C. Kozyrakis, "Reflex: Remote flash ≈ local flash," in *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, p. 345–359.

[26] A. Krizhevsky, "Learning Multiple Layers of Features From Tiny Images," Tech. Rep., 2009.

[27] A. V. Kumar and M. Sivathanu, "Quiver: An Informed Storage Cache for Deep Learning," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, 2020, pp. 283–296.

[28] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston, "Exascale Deep Learning for Climate Analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2018, pp. 649–660.

[29] G. Lee, I. Lee, H. Ha, K. Lee, H. Hyun, A. Shin, and B.-G. Chun, "Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training," in *Proceedings of USENIX Annual Technical Conference*, 2021, pp. 537–550.

[30] Z. Li, B. Jiao, S. He, and W. Yu, "PHAST: Hierarchical Concurrent Log-Free Skip List for Persistent Memory," *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[31] X. Liu, P. He, W. Chen, and J. Gao, "Multi-Task Deep Neural Networks for Natural Language Understanding," *arXiv preprint arXiv:1901.11504*, 2019.

[32] I. Loshchilov and F. Hutter, "Online Batch Selection for Faster Training of Neural Networks," *arXiv preprint arXiv:1511.06343*, 2015.

[33] R. Macedo, C. Correia, M. Dantas, C. Brito, W. Xu, Y. Tanimura, J. Haga, and J. Paulo, "The Case for Storage Optimization Decoupling in Deep Learning Frameworks," in *2021 IEEE International Conference on Cluster Computing*, 2021, pp. 649–656.

[34] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003, pp. 248–255.

[35] Microsoft, "Analyzing and Mitigating Data Stalls in DNN Training," https://github.com/msr-fiddle/DS-Analyzer, 2021.

[36] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and Mitigating Data Stalls in DNN Training," *Proceedings of the VLDB Endowment*, 2021.

[37] H. A. Pierson and M. S. Gashler, "Deep Learning in Robotics: a Review of Recent Research," *Advanced Robotics*, 2017.

[38] C. Pinto, Y. Gkoufas, A. Reale, S. Seelam, and S. Eliuk, "Hoard: A Distributed Data Caching System to Accelerate Deep Learning Training on the Cloud," *arXiv preprint arXiv:1812.00669*, 2018.

[39] S. Pumma, M. Si, W. C. Feng, and P. Balaji, "Scalable Deep Learning via I/O Analysis and Optimization," *ACM Transactions on Parallel Computing*, vol. 6, no. 2, pp. 1–34, 2019.

[40] PyTorch, "PyTorch/Vision," https://github.com/pytorch/vision/tree/master/torchvision, 2021.

[41] C. Seifert, A. Aamir, A. Balagopalan, D. Jain, A. Sharma, S. Grottel, and S. Gumhold, "Visualizations of Deep Neural Networks in Computer Vision: A Survey," in *Proceedings of the Transparent Data Mining for Big and Small Data*, 2017, pp. 123–144.

[42] S. Shi, X. Zhou, S. Song, X. Wang, Z. Zhu, X. Huang, X. Jiang, F. Zhou, Z. Guo, L. Xie, R. Lan, X. Ouyang, Y. Zhang, J. Wei, J. Gong, W. Lin, P. Gao, P. Meng, X. Xu, C. Guo, B. Yang, Z. Chen, Y. Wu, and X. Chu, "Towards Scalable Distributed Training of Deep Learning on Public Cloud Clusters," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 401–412, 2021.

[43] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-Aware Neural Architecture Search for Mobile," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 2820–2828.

[44] L. Wang, S. Ye, B. Yang, Y. Lu, H. Zhang, S. Yan, and Q. Luo, "DIESEL: A Dataset-Based Distributed Storage and Caching System for Large-Scale Deep Learning Training," in *Proceedings of the 49th International Conference on Parallel Processing*, 2020, pp. 1–11.

[45] R. Wang, S. He, W. Zong, Y. Li, and Y. Xu, "XPGraph: XPline-Friendly Persistent Memory Graph Stores for Large-Scale Evolving Graphs," in *IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2022, pp. 1308–1325.

[46] X. Wang, H. Zhao, and J. Zhu, "GRPC: A Communication Cooperation Mechanism in Distributed Systems," *ACM SIGOPS Operating Systems Review*, 1993.

[47] C. Xu, S. Bhattacharya, M. Foltin, S. Byna, and P. Faraboschi, "Data-Aware Storage Tiering for Deep Learning," in *2021 IEEE/ACM Sixth International Parallel Data Systems Workshop*, 2021, pp. 23–28.

[48] C.-C. Yang and G. Cong, "Accelerating Data Loading in Deep Neural Network Training," in *Proceedings of the 26th International Conference on High Performance Computing, Data, and Analytics*, 2019, pp. 235–245.

[49] J. Zhang, H.-F. Yu, and I. S. Dhillon, "Autoassist: A Framework to Accelerate Training of Deep Neural Networks," *arXiv preprint arXiv:1905.03381*, 2019.

[50] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6848–6856.

[51] Z. Zhang, L. Huang, J. G. Pauloski, and I. T. Foster, "Efficient I/O for Neural Network Training With Compressed Data," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, 2020, pp. 409–418.

[52] M. Zhao, N. Agarwal, A. Basant, B. Gedik, S. Pan, M. Ozdal, R. Komuravelli, J. Pan, T. Bao, H. Lu, S. Narayanan, J. Langman, K. Wilfong, H. Rastogi, C.-J. Wu, C. Kozyrakis, and P. Pol, "Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 1042–1057.

[53] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, "Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems," in *Proceedings of the 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2018, pp. 145–156.

[54] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury, "Efficient User-Level Storage Disaggregation for Deep Learning," in *Proceedings of the IEEE International Conference on Cluster Computing*, 2019, pp. 1–12.