SiloD: A Co-design of Caching and Scheduling for Deep Learning Clusters

Hanyu Zhao*† Peking University
Zhenhua Han* Microsoft Research
Zhi Yang‡ Peking University
Quanlu Zhang Microsoft Research
Mingxia Li† USTC
Fan Yang Microsoft Research
Qianxi Zhang Microsoft Research
Binyang Li Microsoft
Yuqing Yang Microsoft Research
Lili Qiu Microsoft Research
Lintao Zhang BaseBit Technologies
Lidong Zhou Microsoft Research

Abstract

Deep learning training on cloud platforms usually follows the tradition of the separation of storage and computing. The training executes on a compute cluster equipped with GPUs/TPUs while reading data from a separate cluster hosting the storage service. To alleviate the potential bottleneck, a training cluster usually leverages its local storage as a cache to reduce the remote IO from the storage cluster. However, existing deep learning schedulers do not manage storage resources thus fail to consider the diverse caching effects across different training jobs. This could degrade scheduling quality significantly.

To address this issue, we present SiloD, a scheduling framework that co-designs the cluster scheduler and the cache subsystems for deep learning training. SiloD treats cache and remote IO as first-class resources and can integrate different state-of-the-art deep learning scheduling policies in a unified scheduling framework. To achieve this, SiloD develops an enhanced job performance estimator to help different schedulers to jointly consider the impact of storage and compute resource allocation while preserving their respective scheduling objectives. The SiloD-enhanced performance estimator leverages the unique data access pattern of deep learning training to develop a closed-form analytic model that captures the diverse cache / remote IO requirements from different training jobs. Evaluations show that SiloD improves the average job completion time, cluster utilization, and fairness by up to 7.4x, 2.57x, and 1.89x, respectively, compared to different combinations of cache systems and cluster schedulers where they operate independently.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Information systems** → **Database management system engines**.

Keywords: Machine learning systems, cloud computing, cache systems

*Hanyu Zhao and Zhenhua Han contribute equally.
†The work is partially done when Hanyu Zhao and Mingxia Li are interns at Microsoft Research.
‡Zhi Yang is the corresponding author (yangzhi@pku.edu.cn).

EuroSys '23, May 9–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567499

ACM Reference Format:
Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Mingxia Li, Fan Yang, Qianxi Zhang, Binyang Li, Yuqing Yang, Lili Qiu, Lintao Zhang, and Lidong Zhou. 2023. SiloD: A Co-design of Caching and Scheduling for Deep Learning Clusters. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3552326.3567499

1 Introduction

As Deep Learning (DL) becomes an increasingly important workload, major cloud platforms now offer services for deep learning training [4, 5, 8, 9]. In the cloud service, deep learning training jobs run on a GPU cluster and read training data hosted in a separate cluster providing storage services like AWS S3 [3] or Azure Blob Storage [6]. Such a setup decouples the storage service from the compute service, leading to a modular and simple solution [39, 44]. However, our experience shows that the remote IO between compute and storage services could become a bottleneck, especially with the introduction of newer generation GPUs (e.g., NVIDIA H100). Therefore, it is beneficial to leverage the local storage in the training cluster as a cache to alleviate the bottleneck.

Currently, the cache subsystem operates independently from DL job scheduling and is unaware of cluster-wide job performance. Meanwhile, state-of-the-art deep learning schedulers are also unaware of the impact from cache subsystems. They focus on arbitrating compute resources (e.g., GPUs and CPUs) with different optimization objectives like job completion time (JCT) [34, 57], fairness [22, 48, 52], or cluster utilization [52, 68]. Such a decoupled design leads to sub-optimal cluster performance. For example, a training job deemed to have high performance by the cluster scheduler may be significantly impacted when limited cache and remote IO become the bottleneck. Such over-estimation results in degraded scheduling quality. This calls for a co-design of cluster scheduling and caching subsystems.

Traditionally, it is non-trivial to design the scheduler and the cache jointly. First, as mentioned earlier, deep learning schedulers have diverse scheduling objectives. An ad-hoc

Dataset size	Year 2020	In 24 months
Task #1	25 TB	100 TB
Task #2	100 GB	1 TB
Task #3	100 GB	3 TB
Task #4	5 TB	10 TB
Task #5	1.5 TB	400 TB

Table 1. The size and growth of datasets for training at Microsoft.

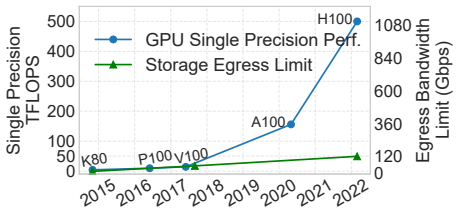


Figure 1. The trend of GPU perf. v.s. egress limits of cloud storage [10, 11].

GPU	Speed	IO
1*V100	1003 images/s	114 MB/s
1*A100	2930 images/s	333 MB/s
8*V100	7813 images/s	888 MB/s
8*A100	16925 images/s	1923 MB/s
1*Gaudi2	5325 images/s	614 MB/s

Table 2. Mixed-precision training and IO speeds of ResNet-50 on ImageNet.

solution to every scheduling policy increases design complexity and is hard to scale. Second, deep learning training exhibits highly diverse performance patterns: different jobs impose different cache and IO demands (§2.2). This further complicates the system design. We have found that even deep learning-aware cache systems could exhibit poor performance (§7) because of caching policies that ignore scheduling impacts.

To address these challenges, we present SiloD, the first deep learning scheduling framework that jointly considers cluster-wide resource allocation for both compute resources like GPUs and storage resources including cache and remote IO. SiloD treats cache and remote IO as first-class citizens and can integrate different state-of-the-art deep learning scheduling policies in a unified scheduling framework.

Despite the diverse objectives, most of the deep learning schedulers require a certain *performance estimator*. Scheduling decisions from different policies are made based on the estimated job performance using different performance estimators. SiloD proposes a general performance estimator to enhance the estimators of existing scheduling policies in a unified way. The SiloD-enhanced performance estimator is then integrated into different scheduling polices and used to make scheduling decisions. In this way, SiloD is able to augment different state-of-the-art deep learning schedulers to jointly perform cache and remote IO allocation while preserving the original objectives of these scheduling policies.

The data access pattern and the computation pattern, despite being highly diverse for different training jobs, are both highly stable and predictable within each individual job. This allows SiloD to derive a unified way of performance estimation by further leveraging the pipelined execution of data loading and computation. The SiloD-augmented performance estimator transforms a joint performance estimation into a two-step process. It first estimates whether data loading will become the bottleneck of the entire training. If so, SiloD will use `IOPerf`, a performance estimator we introduce to analyze the impact of storage to estimate the job performance under IO bottleneck. The `IOPerf` leverages the unique access pattern of deep learning training to derive a closed-form analytical model, given a cache and remote IO allocation. The analytical model is able to capture the highly diverse performance patterns of different training jobs, thus deriving the diverse cache and IO demands. Performance-aware schedulers can leverage the analytical model to exploit the diversity to further optimize their scheduling objectives.

We have implemented SiloD based on Alluxio [46] and Kubernetes [21] with roughly 3,000 and 2,500 lines of code, respectively. Extensive trace-driven experiments on a 96-GPU cluster and a storage service from Azure, as well as high-fidelity simulations, demonstrate that the performance of state-of-the-art scheduling policies for DL training can be greatly enhanced by co-scheduling compute, cache, and remote IO resources. SiloD can improve the cluster performance by up to 7.4x on average job completion time, 2.57x on cluster utilization, and 1.89x on fairness.

In summary, this paper makes the following contributions.

- We propose a unified cluster scheduling framework that extends the state-of-the-art deep learning schedulers to jointly schedule compute and storage resource while preserving the original scheduling objectives.
- We reveal the closed-form relation between job performance and storage resources that can enhance the performance estimator of existing cluster scheduler to accurately predict the impact of compute and storage interaction.
- We implement the state-of-the-art scheduling policies in SiloD's framework, showing significant improvement in job completion time, cluster utilization, and fairness, using real GPU clusters and high-fidelity simulations.

2 Background and Motivation

In this section, we first briefly overview how existing clusters are built for DL training with separated storage and GPU clusters, which could lead to remote IO bottleneck. Existing cache subsystems are inefficient to solve the bottleneck. Then we discuss the unique opportunities and characteristics of DL training that can help to build more efficient clusters by co-designing caching and cluster scheduling.

2.1 Separation of Storage and GPU Clusters

It has become a common practice to build GPU clusters to serve deep learning training workloads. Users submit training jobs to a cluster, and a scheduler queues jobs and dispatches them to the selected servers with allocated GPUs. To achieve high accuracy, a DL model is usually trained with a large amount of data. The size of training data ranges from

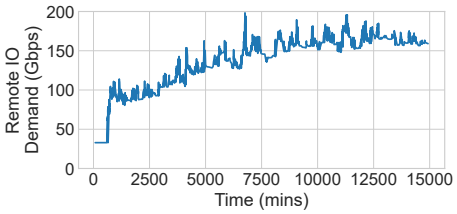


Figure 2. The IO demand of a 400-GPU (V100) cluster running a production trace. The peak IO achieves up to 200 Gbps.

tens of gigabytes to hundreds of terabytes [1, 47, 51, 72]. We have observed increasingly large training datasets for deep learning models. Table 1 shows the sizes of training data reported in early 2020 and how they grow (or planned to grow) in one to two years for five deep learning models, surveyed in production GPU clusters at Microsoft.

With the maturity of cloud storage, users tend to store and manage data in cloud services, such as Blob storage [3, 6] or data lakes [7]. Modern GPU clusters respect this trend by supporting remote storage (e.g., AWS S3 [3], Azure Blob Storage [6], NFS [53]) in a plug-in manner. That is, when submitting a DL job, users can simply specify the storage account and the location of the desired dataset, and the job can directly load the data through remote IO. From the logs collected from production GPU clusters, we observed that around 97.3% of the DL jobs have their data stored in cloud storage. Hence the majority of DL training is executed in a GPU cluster while fetching training data from remote storage; i.e., *separation of storage and GPU clusters* [39, 44].

Remote IO as a bottleneck. Remote data fetching has considerable impacts on the training performance of a deep learning job. The worst case is to read the entire training dataset remotely in each epoch. This imposes a high pressure on the network between the GPU cluster and the remote storage service. Figure 1 summarizes the trend of GPU computation speed (in single-precision FLOPS) and the egress limit of Azure's storage accounts. In the last seven years, GPU speed is improved by 125×, way faster than the 12× increase of egress bandwidth of cloud storage. Table 2 shows that training one ResNet-50 on eight A100 GPUs requires a data loading throughput of 1923 MB/s. Since a large-scale GPU cluster commonly has hundreds or even thousands of jobs running simultaneously, the demand on the total remote IO is prohibitively high. As shown in Figure 2, we have observed the remote IO bottleneck in a medium size production cluster with 400 V100 GPUs (please refer to §7 for the detailed setting). Even the highest supported egress bandwidth of a major cloud storage service shown in Figure 2 (i.e., 120 Gbps, the claimed upper-bound) is far lower than the GPU cluster's aggregated demand if all of the data has to be fecthed from the remote (up to 200 Gbps). The remote IO bottleneck would be even more severe for globally scheduled jobs, whose data may need to be loaded from different regions (e.g., Singularity [59]).

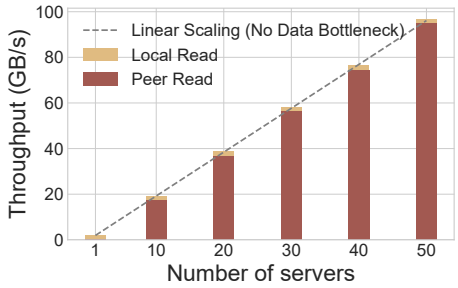


Figure 3. The throughput of distributed cluster running jobs with IO of 1923 MB/s (ResNet-50 on 8 A100s). All datasets are evenly distributed to all servers' cache. In n servers, each job will load $\frac{1}{n}$ data locally and $\frac{n-1}{n}$ data from peer servers.

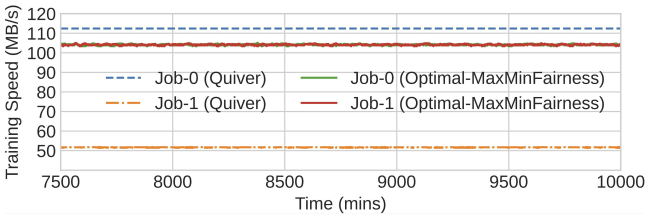


Figure 4. The training speeds of two ResNet-50 jobs training 1.36TB ImageNet-22k. The 2-GPU cluster has 1.4TB cache with 50 MB/s remote IO bandwidth. Quiver spends all cache to Job-0. The optimal max-min fair policy allocates half cache and remote IO to each of the jobs.

The cache subsystem for DL Training. To alleviate the bottleneck, GPU clusters usually leverage local disks of GPU servers to cache a subset of data to reduce the demands to remote IO. There are two types of cache subsystems used by existing GPU clusters.

The first type of cache is built into a data loading library, e.g., CoorDL [50] extends NVIDIA's DALI to support local data caching. The cache is built with the processes of a training job, and is statically allocated with the capacity of a job's local storage inside the running containers/VMs. However, we observe that DL training jobs have diverse demands on cache and remote IO. Some jobs could be over 8000× more efficient when utilizing the cache than others (details are explained in §2.2 and Figure 6). Isolated cache with a static allocation can neither satisfy nor exploit such diversity. A job with slow IO can occupy a large amount of cache, which could be allocated to faster jobs for better efficiency.

The second type of cache subsystem is distributed cache, e.g., Alluxio [46] or Quiver [44], which consolidates the local storage of all cluster servers into a large storage pool shared by all jobs. Modern GPU cluster usually has a high-speed storage fabric (separate from the InfiniBand network used for distributed training) that supports accessing data from peer servers as fast as local disk [54]. Specifically, our experiment in Figure 3 shows, in a cluster of 50 8-A100 servers, jobs can still load data at the throughput of local disks even when

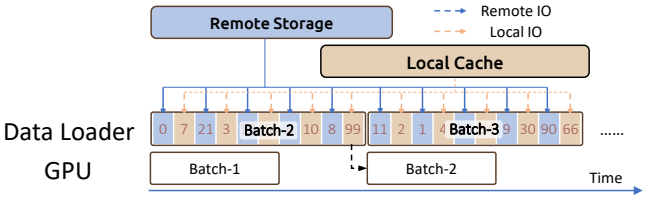


Figure 5. The pipeline of data loading and computation of deep learning training with uniform caching. Each data item has a unique ID. The missed data items are fetched from the remote storage. Because each epoch shuffles the data loading order, the expected cache hit ratio is uniform for all items. The example shows the training has a bottleneck on data loading.

most data is fetched from disks of peer servers. The results show that a distributed cache across the local cluster can generally satisfy the IO demands of training jobs.

However, existing distributed cache systems are built for general workloads [46] and do not exploit the special characteristics of DL training workloads for higher efficiency, e.g., repetitive computation and predictable performance, which are widely utilized by modern GPU cluster management [59, 68, 69]. Moreover, different GPU clusters may have different scheduling objectives, e.g., average job completion time (JCT), cluster throughput, or fairness. Even for distributed cache systems designed for deep learning like Quiver [44], they can degrade the scheduling quality of a job due to the unawareness of the impact to other training jobs in the cluster, missing global optimization opportunities. Figure 4 shows an example where the cluster scheduler needs to optimize for the max-min fairness of job training speed (refer to Equation 8 for formal definition). The cluster has two V100 GPUs, each training a ResNet-50 model with 1.36 TB data. The cluster has 1.4 TB cache and 50 MB/s remote IO bandwidth. Since Quiver is unaware of the scheduling objective of max-min fairness, its aggressively allocates all cache to Job-0, which leads to unfair job performance of Job-1 (114 MB/s v.s. 52 MB/s) (§7.1). However, the optimal max-min fairness policy should allocate the cache and remote IO evenly to the two jobs so that both of them can achieve 107 MB/s training speed.

2.2 Opportunities of Deep Learning Training

Deep learning training exhibits a special data access pattern. A deep learning job trains a neural model by iterating a training dataset multiple times (a.k.a. *epochs*). Within each epoch, the job accesses each data item *randomly* and *exactly once*. The training will shuffle the data access order at the beginning of every epoch. Compared with unpredictable general workloads, deep learning training has unique characteristics that makes it easy to predict the impact of cache.

Uniform caching. Due to the random-and-exactly-once data access pattern, it has been shown that uniform caching is optimal for single training job [44, 50]. By Figure 5 illustrates how a DL training job loads data with uniform caching. In uniform caching, accessed data items are cached until the cache capacity is reached, and will not be evicted thereafter. There is no eviction unless the cache capacity is reduced. Other cache eviction policies like LRU (Least-Recently-Used) may evict useful items, leading to the thrashing issue (please refer to §7.2 for details). When there are multiple jobs in a cluster, uniform caching transforms the cache management from cache eviction problem to a cache space allocation problem. For deep learning training, uniform caching leads to a constant and predictable cache hit ratio w.r.t. the cache capacity (regardless of which items being cached), thus it gives SiloD an opportunity to derive a closed-form analytic model to estimate the impact of cache to a training job.

Diverse cache and IO demands. Despite being a favorable choice for a single job, uniform caching alone does not answer how to allocate cache among multiple deep learning training jobs in a shared cluster. A naïve static cache allocation can lead to undesirable performance. For example, when applying a static cache allocation to CoorDL [50], a uniform-caching-based solution, we observe that CoorDL sometimes performs even worse than LRU (§7.1.2).

Given a cache allocation, different DL training jobs exhibit vastly different behaviors in terms of remote IO savings. In Figure 6, we evaluate 11 jobs on a V100 GPU by observing how much remote IO is saved if we cache the entire dataset of a job while achieving its ideal training speed. We define a job's *cache efficiency* as the amount of remote IO (in MB/s) saved per GB of cache allocated. Denote as f^* and d the IO demand to achieve the ideal training speed and the dataset size, respectively. A job's cache efficiency is $\frac{f^*}{d}$.

Among the 11 jobs, we find ResNet-50 on ImageNet-1k benefits the most from cache: it only costs 143GB to cache all data of ImageNet-1k, saving 114MB/s of remote IO, i.e., a cache efficiency of $\frac{114}{143} \approx 0.8$MB/s/GB. In contrast, BERT on a web search dataset only saves 2MB/s remote IO with 20,971GB cache, i.e., a cache efficiency of 9.5×10^{-5}MB/s/GB. Thus ResNet-50 on ImageNet-1k is more cache-efficient than BERT on Web Search. When these two jobs share a cluster, an efficient cluster scheduler should allocate cache to ResNet-50 to save most remote IOs.

Clearly, a static cache allocation could not take advantage of the diverse cache-efficiency of DL jobs. This observation necessitates the use of an adaptive cache allocation scheme, combined with uniform caching, to exploit the heterogeneous cache efficiency across deep learning training jobs.

Co-designing cache and cluster scheduler. Cache and remote IO could significantly affect the training performance when data loading is the bottleneck, while existing GPU cluster scheduler takes a cache-oblivious approach. This will lead

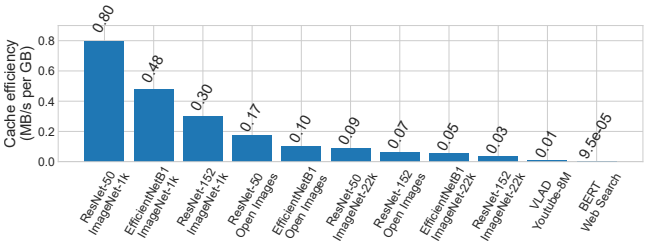


Figure 6. Cache efficiency on a V100 GPU. IO demand to achieve the ideal speed: ResNet-50 (114 MB/s), ResNet-152 (43 MB/s), EfficientNetB1 (69 MB/s), VLAD (10 MB/s), BERT (2 MB/s). The sizes of the datasets are listed in Table 1.

to degraded job performance. For example, the shortest-job-first policy [30, 34] could incorrectly predict the duration of jobs by assuming they do not have IO bottleneck, thus leading to a different order in terms of job completion time. Also, fairness-aware schedulers [48, 52] ignoring the impact of cache could lead to unfairness. Therefore, it is necessary to co-design the cache and cluster scheduler to optimize scheduling objectives accurately. Schedulers can further exploit the heterogeneous cache efficiency to better utilize storage resources according to their scheduling objectives.

3 SiloD Overview

State-of-the-art schedulers for deep learning training jobs are usually performance-aware. They predict job throughput under different compute resource (GPU and CPU) allocation through a performance estimator either based on profiling [68] or prediction models [57]. As shown in [52], the throughput estimator covers widely-used scheduling policies that optimize different objectives under different constraints, including JCT, makespan, max-min fairness [34], finish-time fairness [48]. The main technical challenge is to incorporate the diverse scheduling policies while exploiting the heterogeneous cache efficiency in a unified framework.

SiloD is a system that allocates compute and cache-related resources jointly to training jobs. It is comprised of a scheduler and a cache subsystem: the scheduler allocates compute units (e.g., GPUs), cache size, and remote IO bandwidth (in terms of the egress bandwidth limit of the cloud provider) to each job, while the cache subsystem enforces the cache and bandwidth allocation (i.e., maintain the cached data items and throttles the remote data access of jobs).

Algorithm 1 elaborates how SiloD incorporates scheduling policies to manage compute and storage resources in a unified framework. The scheduling problem can be generally abstracted as allocating cluster resources defined in `totalResource` to `jobs` with the help of a performance estimator `perf(j, R)`. In addition to the compute resources managed by existing schedulers, SiloD further includes storage as first-class resources into `totalResource`.

```
1 def schedule(jobs, totalResource, perf):
2   # perf(j, R): the performance estimator for
      estimating the compute throughput of job j under
      resource allocation R
3   # totalResource: the total resource of the cluster
4
5   SiloDPerf = lambda j, R: min(perf(j,R), IOPerf(j,R))
      # SiloD's enhanced performance estimator to
      jointly consider the impact of compute and storage
       resources
6
7   alloc = Policy.Schedule(jobs, totalResource,
      SiloDPerf)
8   return alloc
```

Algorithm 1. The workflow of SiloD. The underlined variables and functions are introduced/extended by SiloD and the others are inherited from existing schedulers.

Since existing cluster schedulers only consider the impact of compute resource in `perf(j, R)`, SiloD further enhances the estimator `perf` with `SiloDPerf` to estimate the joint impact of compute and storage resources. Observing the pipelined execution of data loading and computation, SiloD transforms the joint performance estimation of a training job into two stages (Line 5). When data loading is the bottleneck, job performance is estimated by `IOPerf`, a SiloD derived closed-form analytic model that estimates job performance given the cache and IO resources. Otherwise, SiloD sticks to the original methods to estimate the computation-bound job performance. We assume `perf` is converted in term of MB/s to match the unit of `IOPerf`. With the help of the enhanced performance estimator, existing multi-resource schedulers can just treat storage resources as yet another resource types whose impact has already been captured by the performance estimator (Line 7). The heterogeneous cache efficiency can be exploited by identifying the impact of different storage resource allocation to their scheduling objectives. This extension is general that applies to any multi-resource schedulers with performance estimator for DL training. Finally, SiloD makes a joint resource allocation that includes compute, cache, and IO bandwidth for each job, and relies on the cache layer to enforce the cache and bandwidth allocations.

Next, we elaborate in §4 how we build the enhanced performance estimator for DL training by leveraging its highly predictable data access pattern. Then, we demonstrate in §5 how existing schedulers can be easily incorporated in SiloD to improve their scheduling objectives, leveraging the enhanced performance estimator.

4 SiloD-Enhanced Performance Estimator

To enhance the performance estimator to estimate the impact of cache to job performance, we analyze the execution pattern of deep learning training and observe that the training executes in a pipeline. As we have shown in Figure 5, the pipeline is executed at the granularity of a batch of data. When a data item (in a batch) is not cached locally, it has to be fetched from the remote storage and consumes remote IO

bandwidth. When the remote IO bandwidth is insufficient, the computation stalls and waits for data loading, leading to the IO bottleneck. Since the data access pattern and computation of each batch are both identical, the training throughput are highly stable and predictable, regardless of whether IO is the bottleneck in the pipeline. This characteristic of DL training is widely adopted by production GPU management solutions [59, 68, 69]. We also leverage this characteristic: the training jobs satisfy the above uniform data access and pipelined execution of computation and data loading when designing SiloD's enhanced performance estimator (we discuss in §6 about how to handle irregular patterns not satisfying them).

Denote as f^* a job's computation throughput when IO is not the bottleneck, i.e., `perf` in Algorithm 1, which is the original estimator used by an existing scheduler. Denote as f the throughput of data loading, i.e., `IOPerf`, which is the estimator for IO given some cache allocation. The end-to-end throughput is then determined by the bottleneck stage, i.e.,

$$\texttt{SiloDPerf} = \min\{f^*, f\}. \tag{1}$$

Due to the discussion in §2.2, SiloD adopts uniform caching for DL training jobs. Denote as c the allocated cache space and d the size of the training dataset. The expected cache hit ratio of a job is simply $\frac{c}{d}$, and the miss ratio is $1 - \frac{c}{d}$. Denote as b the remote IO demand of a job, which equals to the data loading throughput multiplied by the cache miss ratio. Thus, a job's remote IO demand can be calculated as

$$b = f \cdot (1 - \frac{c}{d}). \tag{2}$$

Given the throughput of data loading f, the remote IO demand is determined by the allocated cache size c and the dataset size d. As we have shown in §2.1, the remote IO bandwidth is limited in GPU clusters. Therefore, it requires further allocating (throttling) the remote IO to jobs when the sum of remote IO demand exceeds the bandwidth. In this case, with the cache allocation c and remote IO allocation b, a job's IO throughput f (i.e., `IOPerf`) becomes,

$$f = \frac{b}{1 - c/d}. \tag{3}$$

By combining Equation 3 with Equation 1, the end-to-end training throughput of a job becomes,

$$\texttt{SiloDPerf} = \min\{f^*, \frac{b}{1 - c/d}\}, \tag{4}$$

which can be calculated in a closed-form. This enhanced performance estimator seamlessly integrates the original performance estimator, f^*, with SiloD's new estimation on the impact of cache and remote IO allocation. With `SiloDPerf`, schedulers can model the interaction of computation and data loading so as to manage the compute and storage resources jointly. Our evaluation shows SiloD's enhanced performance estimator is accurate with an error within 3%.

Note that Equation 4 reveals the root cause of the heterogeneous cache efficiency. When a job is fetching data at its ideal throughput (i.e., $f = f^*$), its cache efficiency is exactly the negative derivative of Equation 2, i.e.,

$$\text{Cache Efficiency} = -\frac{\partial b}{\partial c} = \frac{f^*}{d}, \tag{5}$$

which means how much remote IO can be saved by per unit of cache, i.e., what we evaluate in Figure 6. The different computation throughput f^* of different neural model and dataset size d is the sources of the heterogeneity. Schedulers can exploit the heterogeneity in different manners according to their optimization objectives by treating cache and remote IO as first-class resources to be allocated to training jobs.

5 SiloD Policies

In this section, we illustrate how two representative performance-aware deep learning cluster schedulers can be modified by SiloD's enhanced performance estimator to jointly allocate compute and storage resources while preserving their original scheduling objectives (§5.1 and §5.2). For the schedulers that are not performance-aware (i.e., they do not rely on a performance estimator), we use a greedy scheduling policy to exploit the heterogeneous cache efficiency in a best-effort manner without modification to policy (§5.3).

5.1 Multi-resource SJF

Shortest Job First (SJF) is a classic scheduling principle that prioritizes the job with the least duration. It has been extended to consider multiple types of resources in Tetris [30] or support the scheduling of GPUs in Tiresias [34]. In multi-resource SJF, we can unify Tetris and Tiresias by adding GPU as a resource type. Thus, each job will have a performance score defined as its weighted sum of resource demand of all resource types multiplied by its duration, i.e.,

$$\text{score} = \min_{\mathbf{R}} \sum_t w_t \cdot R_t \cdot \underbrace{\left(\frac{j.\text{numSteps} \cdot j.\text{stepDataSize}}{perf(j, \mathbf{R})}\right)}_{\text{job duration}}, \tag{6}$$

where w_t is the weight of the t-th resource type, $\mathbf{R}$ is a vector of allocation of all resource types, and R_t is the allocation of the t-th resource type in $\mathbf{R}$, j.numSteps is the job j's total number of steps and j.stepDataSize is the size of data consumed per step. The jobs with the least score will be scheduled first by the multi-resource SJF policy.

For SiloD to incorporate multi-resource SJF, $\mathbf{R}$ includes cache and remote IO as another two types of resources in addition to compute resources. The performance estimation function $perf(j, \mathbf{R})$ is simply replaced by Equation 4 to include the impact of cache and remote IO. After the replacement, Equation 6 becomes

$$\text{score} = \min_{\mathbf{R}} \sum_t w_t \cdot R_t \cdot \left(\frac{j.numSteps \cdot j.\text{stepDataSize}}{\texttt{SiloDPerf}(j, \mathbf{R})}\right). \tag{7}$$

In our evaluation in §7, the weight w_t is set to $\frac{1}{totalResource[t]}$ according to [30]. Note that, there is a trade-off in Equation 7: higher resource allocation would increase **R** but reduce the job duration $j.numSteps/\texttt{SiloDPerf}(j, \mathbf{R})$. The policy would implicitly benefit jobs with higher cache efficiency because they can run faster with less resources. For example, two ResNet-50 jobs using the same number of GPUs and number of steps, could consume very different cache if one trains ImageNet-1k (143GB) and the other trains ImageNet-22k (1.3TB) thus the former should be first scheduled.

5.2 Gavel

Gavel [52] is the state-of-the-art scheduler for DL training that supports heterogeneous resources. It generalizes a wide range of existing scheduling policies to schedule training jobs by solving mathematical programming. We use its default objective, max-min fairness, as the example to demonstrate how we incorporate it in SiloD. The vanilla programming in Gavel's max-min fairness is formulated as follows:

$$\max_{R} \min_{j} \frac{perf(j, R[j])}{perf(j, R^{equal})} \quad (8)$$
$$s.t.\ Sum(R) \leq totalResource,$$

where $R[j]$ is the resource allocated to job j and R^{equal} is the equal resource division among all jobs. The max-min fairness objective maximizes the job with the least performance improvement over the equal resource division. Ignoring cache and remote IO could lead to inaccurate allocation that sacrifices the performance of the least fair job. For example, when the least fair job has a high ideal training throughput but actually has an IO bottleneck due to low cache efficiency, the vanilla Gavel could overestimate its performance and fail to optimize its objective. Also, it could waste more compute resources on jobs with IO bottlenecks, leading to GPU underutilization. Similar to multi-resource SJF, our extension adds cache and remote IO as two new dimensions in R and replaces $perf(j, R)$ with $\texttt{SiloDPerf}$. After the replacement, the optimization problem of Equation 8 becomes

$$\max_{R} \min_{j} \frac{\texttt{SiloDPerf}(j, R[j])}{\texttt{SiloDPerf}(j, R^{equal})} \quad (9)$$
$$s.t.\ Sum(R) \leq totalResource.$$

This extension can not only support the max-min fairness objective but also all other objectives supported by Gavel.

5.3 A Greedy Policy for All Schedulers

The insight of $\texttt{SiloDPerf}$ not only enables SiloD to incorporate performance-aware schedulers, its implication to cache efficiency also allows SiloD to work with schedulers that do not rely on a performance estimator (e.g., FIFO). To this end, we propose a greedy policy that is less intrusive to an existing cluster scheduling architecture. This policy allows SiloD to be deployed as a standalone component without modifying existing cluster schedulers. Since the cluster scheduler is not aware of potential IO bottleneck when allocating compute resources, the greedy policy minimizes the remote IO consumption in a best-effort manner so that the impact of IO to original scheduling objectives can be minimized.

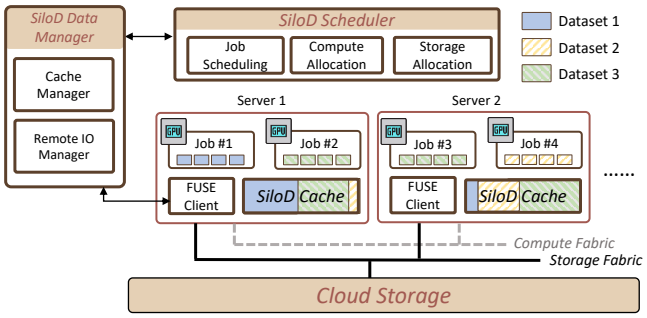


Figure 7. SiloD architecture.

The greedy policy directly leverages the cache efficiency we derived in Equation 5 to maximize the remote IO reduction by allocating more cache to the most cache-efficient jobs. Again, the policy assumes the ideal throughput of a job f^* (when IO is not a bottleneck) can be profiled offline.

Algorithm 2 elaborates the pseudo-code of the greedy policy. Each job first calculates its cache efficiency with Equation 5 (Line 1-2). The datasets with the highest cache efficiency are first cached until the cache space is full (Line 3-5).

Algorithm 2 Greedy cache allocation policy

```
1: for job j in all jobs do
2:     j.CacheEfficiency = j.f* / j.datasetSize
3: for job j in descending order of j.CacheEfficiency do
4:     alloc.Cache[j] = min(j.datasetSize, totalCache)
5:     totalCache -= alloc.Cache[j]
6: return alloc
```

6 Implementation and Optimizations

SiloD designs a simple interface between the cluster scheduler and the distributed cache system. As shown in Figure 7, SiloD is comprised of two major components: (1) *SiloD Data Manager* serves in the storage layer to enforce the allocations made by the scheduler; and (2) *SiloD Scheduler* extends the responsibility of the compute-only resource scheduler from job scheduling (i.e., GPU allocation) to compute-storage joint allocation. We implement SiloD Data Manager by enhancing Alluxio [46], a distributed caching system, with roughly 3,000 lines of Java code. We replace Alluxio's default caching policy with uniform caching and control the caching and data fetching using Alluxio's low-level IO operation APIs. We implement SiloD Scheduler and scheduling policies on Kubernetes [21] with 2,500 lines of Python and Go code.

Allocation APIs
void allocateCacheSize(dataset_uri, cache_size)
void allocateRemoteIO(job_id, io_speed)

Table 3. The APIs for storage resource allocation.

SiloD data manager. SiloD's data manager provides new capabilities to the scheduler by exposing allocation interfaces listed in Table 3. A cluster scheduler uses the interface to allocate cache and remote IO to two types of entities: *job*s and *dataset*s. Remote IO is allocated to jobs directly, while cache is allocated to datasets, and to the associated jobs indirectly. Multiple jobs can transparently share the same cache space for the same dataset. In contrast, since jobs using the same dataset still read data items in different order, remote IO is exclusive to each job. The awareness of the concept of dataset and job differentiates SiloD from traditional cache systems built on the notion of files and data blocks.

SiloD data manager sets up FUSE (Filesystem in USErspace) clients co-located with training tasks to manage the cache, throttling remote IO and maintaining the metadata of datasets on each server. When enforcing the cache allocation, the data manager determines if each data item fetched from the remote can be cached according to if the allocated cache size has been reached. If a dataset exceeds the cache allocation, SiloD data manager implements the uniform caching discussed in §2.2. When a job's allocated cache is reduced, Data Manager will evict its cached data items randomly, which still satisfies the uniform access pattern of uniform caching. When the data items are fetched from the remote storage, the FUSE client at each server will throttle the fetching speed to ensure it follows within the job's remote IO allocation. For distributed data-parallel training, the remote IO allocation is equally distributed to each worker of the job. Our implementation of SiloD is non-intrusive to DL training frameworks (e.g., Tensorflow and PyTorch). The dataset is mounted in a folder by FUSE thus jobs can read data directly with POSIX APIs. The job throughput and progress in each epoch can be monitored via data access requests.

Dataset sharing. In a cluster where multiple jobs can share the same dataset, the scheduling policy needs to be aware of it to maximize the benefit of sharing. In SiloD, the policies discussed in §5 manage cache at the dataset level. We separate the charging of cache consumption and the impact of cache allocation in `SiloDPerf` from other resources, i.e., the cache consumption is charged by only once for each dataset instead for every jobs. Multiple jobs sharing a dataset can benefit from the same allocated cache. For example, two jobs using ImageNet-1k only consume 143 GB cache to cache all data instead of 143*2 = 286 GB. Moreover, in Algorithm 2, the cache efficiency is defined at dataset-level, which is the sum of all jobs' cache efficiency using the same dataset. The datasets with the highest cache efficiency are first cached.

Delayed effectiveness of cache. An interesting phenomenon of cache for DL training is the delayed effectiveness of the newly cached data items. Since DL training reads each data item exactly once per epoch, any newly cached data items will never be accessed again until the next epoch. Even though the newly cached item consumes the cache space, but it does not help to reduce the remote IO until the next epoch. Therefore, accurate estimation of job performance should use the effective cache size. However, since multiple jobs may use the same dataset, it is unknown beforehand if a newly cached item by one job is effective or not for other jobs. This could make scheduling difficult since the impact of cache allocation becomes unpredictable. Fortunately, the delayed effectiveness only has a limited impact that lasts for at most one epoch for newly cached items. A DL job usually trains a model for tens of epochs, thus for most of the time, the cached data are effective. In Figure 8, our trace-driven experiment also shows that, on average, the cluster has over 91.7% of cached data are effective. It will have little impact on the cluster efficiency even scheduling policies in SiloD do not explicitly handle the delayed effectiveness. SiloD also supports fine-grained management for policies to inspect the effective cache size and the instantaneous remote IO demand, by maintaining a bitset for each job to track its accessed items.

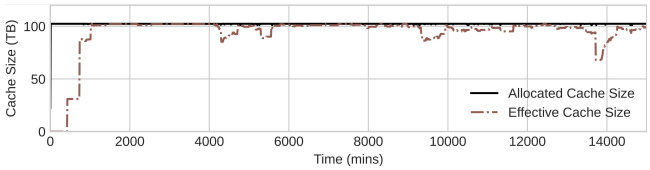


Figure 8. The effective cache size v.s. the allocated cache size in our trace-driven experiments. On average, over 91.7% of cached data are effective.

Handling irregular data access. SiloD's enhanced performance estimator makes two assumptions: (1) each data item is read exactly once per epoch with a *uniform* probability and (2) the computation and data loading form an optimized pipeline (i.e., satisfying the bottleneck assumption). These assumptions hold for most deep learning training jobs but may be broken for specific training strategies. When a cluster is mixed by regular jobs satisfying SiloD's assumptions and irregular jobs, we partition the cache and remote IO into two parts for all regular jobs and irregular jobs, respectively. We allocate resources to the regular jobs in the first partition still using `SiloDPerf`. The irregular jobs in the second partition fall back to the original scheduling policy and estimator, and share the cache and remote IO within the partition. In this way, the regular DL jobs can still benefit from exploiting the heterogeneous cache efficiency without being impacted by potential anomalies due to mis-estimation of irregular jobs. Our evaluation in §7.4 demonstrates how SiloD performs under a special training paradigm, curriculum training [35],

that sorts the data items by training "difficulty" but not uniform probability.

Fault tolerance. SiloD is fault-tolerant. The SiloD Scheduler, SiloD Data Manager, and each FUSE Client on each server are deployed as Kubernetes StatefulSet to ensure each of them is deployed as a single running instance. SiloD Data Manager maintains the allocation decisions and cache status of each server in memory, which is not persistent. The allocation of remote IO and cache is stored in "pod annotation" for the pods of each job, which is kept reliably by Kubernetes. For the job with multiple pods, the remote IO allocation is proportionally divided to each pod and the cache allocation is same for all pods. When SiloD Data Manager recovers from crashes, it reconstructs the status by collecting the information from pods. When the FUSE client restores from crashes, it pulls the allocation and metadata of running pods on the server. The cache content on each server is stored on local disk thus can be reliably restored when the server restarts. SiloD does not add stateful information into the cluster scheduler, thus their fault tolerance is handled by their original approach.

7 Evaluation

In this section, we evaluate SiloD with various settings in real GPU clusters and large-scale simulations. We compare SiloD with three state-of-the-art cache solutions commonly used in production DL clusters with three representative scheduling policies. Overall, SiloD improves the JCT, makespan and fairness by up to 7.4x, 2.57x and 1.89x, respectively.

GPU Acceleration. To evaluate SiloD in a large-scale cluster of fast V100 GPUs with a lower cost, we design an approach to accelerating the training on a K80 GPU cluster to investigate the data loading performance of running the same trace in a V100 GPU cluster. In the experiment, we first profile the training speed of selected models on real V100 GPUs. Then, we execute the same model on K80 GPUs by processing the same training pipeline of data loading, preprocessing and model aggregation, but replacing the model execution (forward pass and backward pass) with "sleep()" for the profiled duration from V100. Since deep learning training usually has a very stable mini-batch duration [68], the IO behaviour in accelerated K80 GPUs is almost the same as real training of V100 GPUs. Our micro-benchmark in §7.1.1 shows this GPU acceleration approach has a very high fidility with up to 3.2% error on the average JCT and 3.7% on makespan.

Workloads. Eight popular deep learning models are selected from GitHub together with open datasets and our internal production dataset, which are listed in Table 4. All datasets are stored in Azure blob storage. All GPU VMs are provisioned by Azure. In a small-scale experiment on 8 V100 GPUs, we use a simple workload composed by ResNet-50, EfficientNetB1, and BERT to demonstrate how different solutions behave. In the large-scale 96-GPU cluster, we generate a trace following the the same job duration distribution reported by Microsoft [41], which includes single-GPU training and distributed multi-GPU training. The total number of training steps of a job is set by multiplying the throughput on V100 (in steps/sec) by the duration of the job, which follows the process of previous works, e.g., Gandiva [68] and Gavel [52]. Because the production GPU cluster has more datasets and diverse combinations of models and datasets than Table 4, we maintain the the diversity by assuming all jobs use different datasets. In §7.3, we set the portion of jobs sharing the same dataset to evaluate the benefit of dataset sharing.

Dataset	Size	Model
ImageNet-22k [24]	1.36 TB	AlexNet [43], EfficientNetB0 [64], EfficientNetB1 [64], InceptionV3 [62], ResNet-50 [36], ResNet-152 [36]
Open Images [2]	660 GB	
ImageNet-1k [24]	143 GB	
Youtube-8M [13]	1.46 TB	VLAD [40]
Web Search	20.9 TB	BERT [25]

Table 4. Dataset and models used in the evaluation.

Baselines. We compare SiloD with three representative cache solutions used in production GPU clusters.

1. Alluxio [46]: a distributed cache system for general workloads, including data analytic and machine learning. We use its default policy LRU that always evict the least recently used data from the cache;
2. CoorDL [50]: a data-loading library for DL training. Each job caches data independently and uniformly using the local disks inside its VM (e.g., 368GB per V100 in Azure [9]);
3. Quiver [44]: a distributed cache system for DL training to optimize the data loading latency. We use its policy for comparison that preferentially assign cache space to datasets the highest ratio of benefit-to-cost (i.e., the ratio between latency reduction and the cache consumption).

We use three scheduling policies for the evaluation: FIFO, SJF, and Gavel (max-min fairness) [52]. The FIFO policy is evaluated in the real cluster experiments (including the micro-benchmark and the 96-GPU experiment). SiloD follows the scheduling order set by the FIFO policy and allocates cache/remote IO for the scheduled jobs. In the simulations, we evaluate all of the three scheduling policies and SiloD can jointly decide the scheduling order and storage resource allocation in SJF and Gavel.

7.1 Real Cluster Experiments

7.1.1 Micro-benchmark with V100 GPUs.

We design the micro-benchmark with two purposes. First, it is used to evaluate the fidelity of the proposed GPU acceleration approach and our simulator used in §7.2. Second, it helps us to understand how different solutions allocate cache resources in a small cluster with diverse IO demand.

	Evaluated in this paper			Production
Cluster	8 V100	96 K80	400 V100	~1900 V100
Remote IO limit	1.6 Gbps	8 Gbps	32 Gbps	120 Gbps

Table 5. The remote IO limit in evaluation (scaling down from our production cluster according to the cluster size).

	Average JCT (relative error)		
	Real V100	Accelerated K80	Simulation
SiloD	**3366**	3339 (0.7%)	3403 (1.1%)
CoorDL	**4278**	4328 (1.1%)	4406 (3.0%)
Alluxio	**4378**	4519 (3.2%)	4484 (2.4%)
Quiver	**3609**	3534 (2.1%)	3592 (0.4%)

	Makespan (relative error)		
	Real V100	Accelerated K80	Simulation
SiloD	**3807**	3747 (1.5%)	3718 (2.3%)
CoorDL	**4870**	4925 (1.1%)	4918 (0.9%)
Alluxio	**5080**	5272 (3.7%)	4986 (1.8%)
Quiver	**3933**	3767 (4.4%)	3915 (0.4%)

Table 6. Average JCT and makespan (in minutes) in the 8-V100 experiment (bold), and relative error using the acceleration approach and the simulator.

In the micro-benchmark, we use eight V100 GPUs (two 4-V100 VMs). Each VM has 1TB SSD available as the cache. As shown in Table 5, we scale down the remote IO bandwidth to 1.6 Gbps (200 MB/s) from the production-scale cluster to match its small cluster size. We build a trace of 5 jobs: four 1-GPU image classification jobs (two for ResNet-50, two for EfficientNetB1), and one 4-GPU BERT job. The four image classification jobs use four different datasets, each of which has 1.3TB synthesized images. The 4-GPU BERT job trains our internal web search data of 20.9 TB. For the jobs, we configure the numbers of epochs to let them run for an acceptably long period (~3,500 minutes) under the ideal throughput. This translates to 13 epochs for ResNet-50, 10 epochs for EfficientNetB1, and 0.07 epochs for BERT. Note that BERT has a very big dataset, thus it does not complete an epoch within the period and cannot benefit from cache (since it also has a very low demand for cache (as analyzed below), we believe this does not affect our main conclusion).

Table 6 shows the average job completion time (JCT) and makespan (the completion time of the last job, which is a metric of cluster utilization) of SiloD and the three baseline systems. We also executed the same jobs on the accelerated K80 GPUs and in the simulator. The relative error is within 3.2% and 4.4% for the average JCT and makespan, respectively, which proves both our acceleration approach and simulator have a very high fidelity.

Figure 9 demonstrates the variation of total job throughput. Before the 460-th minute, all systems have the same performance because the cached items have not become effective in the first epoch. At the 460-th minute, the four

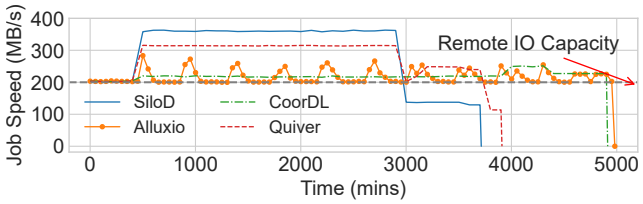


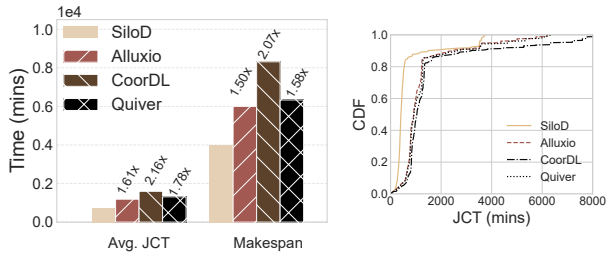
Figure 9. The time-varying total job throughput in the 8-V100 experiment.

image classification jobs enter the second epoch thus the cached data become effective. The figure clearly shows that SiloD achieves the highest cache efficiency. In fact, this is also the optimal cache efficiency: all jobs in SiloD do not have bottleneck on data loading. This is achieved by allocating the cache to the most cache-efficient jobs, i.e., ResNet-50, which has a better cache-efficiency ($\frac{114MB/s}{1.3TB}$ = 87 MB/s/TB) than EfficientNetB1 ($\frac{69MB/s}{1.3TB}$ = 53 MB/s/TB) and BERT ($\frac{8MB/s}{20TB}$ = 0.4 MB/s/TB). In the 2 TB cache, one ResNet-50 caches all its 1.3 TB data. The other ResNet-50 uses the rest 700 GB and hence still needs ~ 52.6 MB/s remote IO. The rest three jobs load data directly via remote IO (two EfficientNetB1: $69 * 2$ MB/s and one BERT: 8 MB/s). Thus, 200 MB/s remote IO is enough for all jobs to achieve their ideal training speed. Finally, the total job throughput decreases gradually after the jobs complete.

Since CoorDL as a data-loading library that independently caches data for each job. It is unaware of the cache efficiency of jobs, thus wastes half of the total cache capacity (1TB) on BERT. Alluxio's LRU often evicts cached items that have not been read in the running epoch thus leads to the cache thrashing: the probability of a useful item being evicted decreases over time within one epoch. This is why we obverse jobs' real throughput in LRU is jagged over time. Quiver's greedy policy performs slightly worse than Algorithm 2 of SiloD because it only caches the images of one of the ResNet-50 jobs (1.3TB) and wastes the rest 0.7TB cache space. According to Quiver's claim, jobs do not benefit from Quiver if it cannot entirely fit into the cache. However, according to our observation in Equation 4, a job can still benefit from SiloD's cache even when its dataset is partially cached.

7.1.2 Experiment in a 96-GPU Cluster.

In this section, we use the proposed GPU acceleration approach to accelerate 96 K80 GPUs to evaluate the performance in a cluster of 96 V100s. As noted in Table 5, we use a remote IO bandwidth of 8 Gbps (1 GB/s) for the 96-GPU cluster, which is scaled down from our production cluster. Figure 10a shows the average JCT and makespan of all systems. SiloD improves the average JCT by up to 2.16x and the makespan by up to 2.07x. Figure 10b shows the distribution of JCT. It shows SiloD is constantly better than the two baselines, which means its improvement is due to higher cluster efficiency but not scarifying certain types of jobs. Figure 11



(a) JCT and Makespan **(b)** JCT Distribution

Figure 10. The average JCT, makespan and JCT distribution of the four policies in the FIFO-scheduled 96-GPU cluster.

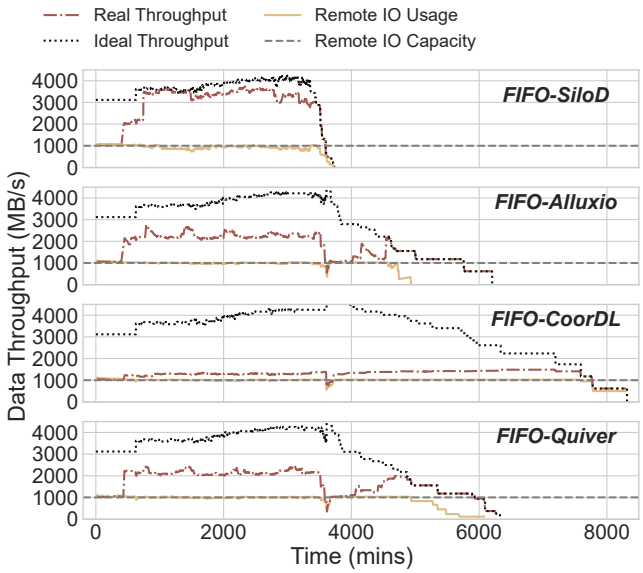
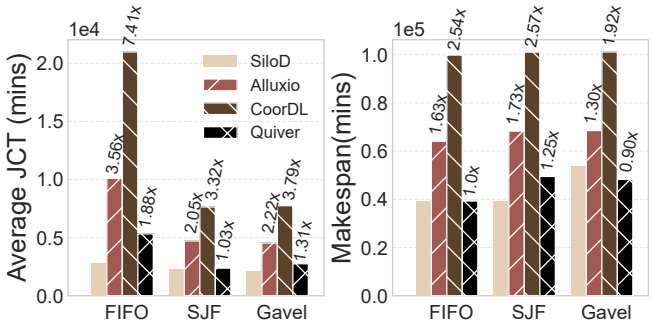


Figure 11. The total remote IO consumption, ideal training throughput, real training throughput in the 96-GPU cluster.

illustrates more details of how data throughput varies over time. SiloD has the highest cache efficiency and achieves a throughput close to the ideal throughput.

CoorDL benefits the least from cache. It only saves at most 490 MB/s remote IO (the gap between the lines of Remote IO Capacity and the Real Throughput in Figure 10a). Although Alluxio's LRU policy is inefficient for a single job (due to cache thrashing), it performs better than CoorDL from the perspective of the overall cluster when there are a lot of jobs. The fast jobs that consume more IO will have a higher probability to evict the data of slow jobs that consume less IO. When these fast jobs are more cache-efficient (although not always), LRU would indirectly benefit them by storing more items for them. This makes Alluxio more efficient than CoorDL by exploiting the heterogeneous cache efficiency.

Quiver performs closely to Alluxio, whose JCT is 1.78x higher than SiloD. In addition to the wasted cache space due to not supporting partial caching, Quiver uses the performance (in terms of latency) profiled online to estimate the benefit of caching a dataset, which is not stable when the



(a) Average JCT **(b)** Makespan

Figure 12. The performance of FIFO, SJF, Gavel using SiloD, Alluxio, CoorDL and Quiver in the 400-GPU simulation.

remote IO fluctuates. We observe Quiver sometimes wrongly evicts effective data from cache and had to rebuild the cache with one more epoch due to the unstable caching priority due to profiling. Instead, SiloD uses the closed-form Equation 8 to calculate the benefit that depends on stable throughput: a job's ideal training speed and its dataset size that can be obtained robustly offline.

7.2 Large-scale Simulations

To evaluate SiloD's performance in a larger cluster and with a longer trace, we build an event-driven simulator with $\sim$ 5,200 lines of Go code that simulates the job training behaviors in the granularity of mini-batch. The simulated events include job submission, the start and finish of a job, the start and finish of the IO of a mini-batch, the start and finish of training a mini-batch on GPU. The duration of remote IO event is determined by the remote IO allocation.The duration of training a mini-batch is the same as the time we profiled on real V100 GPUs. We also execute the fidelity test for the simulator in the real 8-V100 and the 96-GPU clusters. The errors of JCT and makepsan are only up to 5.7% and 8.5%, respectively. The errors are mainly caused by the accumulated launching overhead of short jobs, which are difficult to capture accurately in the simulator.

We simulate a cluster of 400 V100 GPUs. As shown in Table 5 we use a 32 Gbps (4 GB/s) remote IO limit, which is 4× that of the 96-GPU cluster. The trace is constructed similarly to the 96-GPU experiment in §7.1.2, but with more jobs and longer running time ($\sim$ 4 weeks). In SiloD, we evaluate all the scheduling policies we introduced in §5. We also evaluate the vanilla version of these three policies on Alluxio, CoorDL and Quiver without the extension.

Figure 12 shows the average JCT and makespan of the three scheduling policies on the four cache solutions. SiloD achieves up to 7.4x improvement on the average JCT and 2.57x improvement on the makespan. The speedups in the large-scale simulation are higher than the real cluster experiments in §7.1.2, because the queue in the 4-week trace builds up more extremely with the longer trace. Compared to clusters without co-design, SiloD can further exploit the

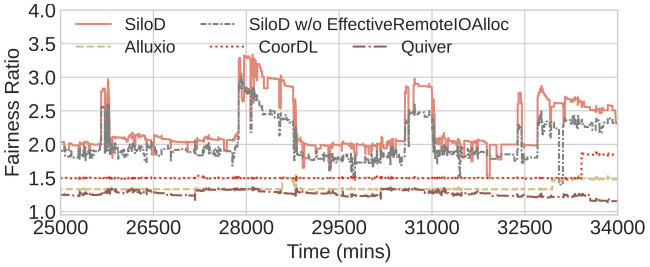
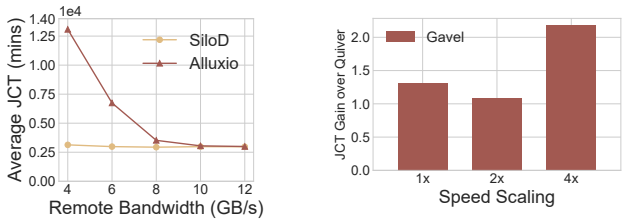


Figure 13. The fairness ratio over time in the 400-GPU clusters scheduled by Gavel. The higher the better.



(a) Remote Bandwidth **(b)** GPU speed

Figure 14. The impact of remote bandwidth and GPU speed.

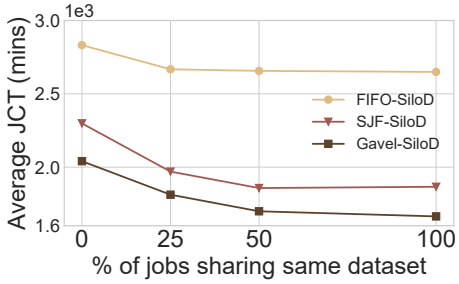


Figure 15. Impact of dataset sharing

heterogeneous cache efficiency in SJF and Gavel because of the integration of SiloD-enhanced performance estimator into the scheduling policy. SiloD can even outperform the deep learning-aware caching system, i.e., Quiver, by up to 1.31x on makespan and 1.25x on average JCT. Note that in Figure 12b shows Gavel on Quiver achieves slightly lower makespan than SiloD. This is because Gavel optimizes for fairness, instead of makespan, which we elaborate next.

Recall that Gavel uses the max-min fairness in Equation 8 as the scheduling objective. With the SiloD-enhanced performance estimator, the fairness in Gavel is further improved. Figure 13 demonstrates the fairness ratio, i.e., the objective of Equation 8, over time of the four storage systems running with Gavel scheduler. SiloD achieves the highest fairness ratio due to the co-scheduling directly optimizes the fairness objective. The baselines schedulers are unaware of the impact of storage thus may lead to "unfair" decisions to sacrifice the performance of the jobs with the the least performance gain over equal-sharing. Overall, the average fairness ratio of SiloD, CoorDL, Alluxio, Quiver are 2.56, 1.51, 1.39 and 1.35, respectively. SiloD improves the fairness by up to 1.89x. It demonstrates why cluster schedulers should consider the impact of storage when optimizing their objectives.

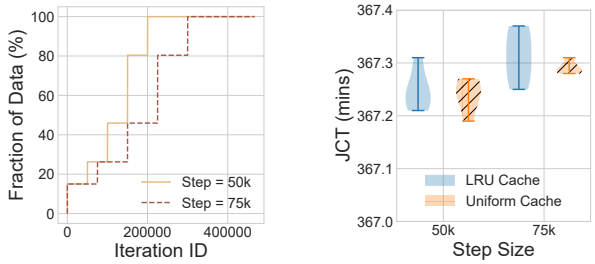
Furthermore, we evaluate the case of disabling the allocation of remote IO to show the necessity of managing both cache and remote IO. In this case, the remote IO is typically allocated by the cloud provider. We use a simple fair share algorithm for remote IO in the simulation. We observe that the average JCT and makespan do not change much (< 2%) but the average fairness ratio is significantly degraded by 31% to 1.94. This shows that controlling both resources instead of cache only is necessary for metrics like instantaneous fairness.

Impact of Bandwidth. SiloD is proposed to solve the remote IO bottleneck. Therefore, in Figure 14a, we vary the remote IO bandwidth to study its impact. Under the FIFO scheduler, we compare SiloD with Alluxio since it represents the most commonly-used off-the-shelf system setting. As we have observed previously in Figure 12, SiloD performs much better than the baselines when remote IO is limited. With the increased remote IO, SiloD benefits less from its IO-aware resource allocation. When remote IO bandwidth has 10 GB/s, even Alluxio using the LRU cache will not have the bottleneck on remote IO thus leading to the same average JCT as SiloD. In clusters with limited remote IO bandwidth, SiloD's approach should be considered to utilize the cache in the most efficient manner.

Impact of GPU Speed. As we have shown in Figure 1, modern GPUs are getting faster. To understand the impact of faster GPU speed, we scale the GPU speed by a factor in Figure 14b. We compare SiloD with the best performing baseline, Quiver, using Gavel scheduler. With faster GPUs, the IO demand for data loading increases, making jobs easier to meet IO bottleneck. SiloD gains 2.17× speedup over Quiver on the fastest GPUs with a 4× speed scaling. This is because SiloD jointly allocates compute and storage resources by considering its bottleneck. With increased GPU speed, SiloD observes more jobs becoming IO bottleneck thus allocating more cache to them to maintain the "max-min fairness". But Quiver's greedy policy still allocates cache space unfairly that starves some jobs severely. This further proves the necessity to manage compute and storage resources in a unified manner.

7.3 Benefit of Dataset Sharing

In previous experiments, we assume the datasets of all jobs are different so that we can obtain a cluster of diverse datasets similar to production clusters. However, SiloD's design naturally supports different jobs to share the same dataset. To study the impact of dataset sharing, in Figure 15, we vary



(a) Pacing function **(b)** Uniform v.s. LRU

Figure 16. The impact of curriculum learning

the percentage of jobs from the trace that can share their datasets. With more jobs sharing their datasets, the average JCT decreases due to the improved cache efficiency. When jobs with the same dataset can be shared, there is no bottleneck on remote IO and the overall JCT can be improved by 22% for SJF and Gavel. But we observe FIFO-SiloD's performance without dataset sharing achieves close to the optimal performance of the fixed scheduling order, thus sharing all datasets only brings 6.9% improvement. This also proves the benefit of co-scheduling compute and storage resources, e.g., the scheduler may schedule jobs with the same dataset together to actively exploit higher cache efficiency.

7.4 Curriculum Learning

By default, SiloD uses uniform caching based on the assumption of data access pattern. This may not fit all training paradigms. For example, curriculum learning has a different data access pattern [35]: it first sorts the data items based on its learning difficulty. Then, each batch will *uniformly* sample the items within data defined by a pacing function, which do not have the concept of "epoch". The assumption that an item is only accessed once per epoch is not valid for curriculum learning. Also, those easy samples can be used more often than hard samples, because the hard samples are only presented at the latter stage of a training. Figure 16a demonstrates two pacing functions that defines which data can be used at the i-th iteration. We adopt the commonly-used exponential pacing function [35], which is defined as follows,

$$g(i) = min(starting\_percent \cdot \alpha^{\lfloor \frac{i}{Step} \rfloor}) \cdot N, \qquad (10)$$

where $starting\_percent$ defines the initial data presented to the network, α defines the growth speed, and $Step$ defines how frequently the training will present new data. In Figure 16b, we compare the performance of Uniform Cache and LRU Cache on curriculum learning. We evaluate both $50k$ and $75k$ as the step size to train ResNet-50 on ImageNet-22k. We find LRU cache and Uniform cache have the similar JCT ($\sim$ 367 mins.) on the two step sizes. Each setting is repeated by 5 times. The job completion time (JCT) is shown in Figure 16b. LRU performs as good as Uniform cache and no longer suffers from thrashing, because the newly cached items will be immediately effective and could be used in the next batch. Since the data items presented by the pacing function have the same probability to be sampled to form a batch, the expected job throughput derived in Equation 4 still holds for both LRU and Uniform cache, and is not affected by the pacing function.

8 Related Work

Storage systems for deep learning. Recent works proposed various caching solutions to alleviate the data bottleneck for DL training jobs [44, 50, 58]. The key difference between prior works and SiloD is that the former solutions are still designed as an independent storage system, while SiloD is a co-design of cluster scheduler and cache that improve end-to-end job performance by jointly and efficiently allocating compute and data-related resources.

CoorDL [50] is a library to accelerate IO from disks to memory and the preprocessing on CPUs. CoorDL shares the same insight with SiloD that one should never replace cached data items within a single training job, and adopts the same uniform caching as SiloD. However, the caching in CoorDL is limited to single job as a data-loading library. It does not consider cache allocation across multiple jobs, which has been shown critical to GPU cluster and job efficiency in our evaluations. SiloD unleashes the potential of uniform caching by exploiting the heterogeneous cache-efficiency via global allocation and a co-design with cluster scheduler.

Quiver [44] is a distributed cache optimized for deep learning training. It also leverages the data access pattern of deep learning training to optimize cache allocation but only at the caching layer. Quiver adopts a policy to allocate cache to the most beneficial jobs based on online profiling. In contrast, SiloD advocates a co-design of caching and cluster scheduling. The cache allocation is delegated to the cluster scheduler to be jointly managed with compute resources in a unified framework. Schedulers can leverage SiloD's enhanced performance estimator (offline profiling and calculation) to optimize their respective scheduling objectives at the cluster level. Our evaluation shows the co-designed framework can bring substantial benefit over cache-only solutions on job completion time, cluster efficiency, and fairness.

There are also other works targeting data loading, preprocessing, caching, and their intersections for deep learning training jobs, which are complementary to SiloD. Revamper [45] caches preprocessed data at certain stages during the preprocessing pipeline to reuse it. Cachew [32] proposes autocaching and autoscaling policies for data preprocessing. OneAccess [42] also targets the dataset sharing scenario and allows different jobs to share data IO and preprocessing to reduce data overhead. DIESEL [67] and AIStore [15] pack small files to reduce small-file access overhead. Petastorm [33] use a column-wise format to reduce IO when a job only needs part of the columns of the data.

Scheduling systems for deep learning. Existing deep learning schedulers focused on compute resources (GPUs) and studied how to allocate GPUs efficiently to improve various metrics such as GPU utilization [68, 69], JCT [34, 57], fairness [22, 48, 52, 71]. SiloD shows that being agnostic to storage system these schedulers leads to suboptimal scheduling decisions that are inconsistent with their goals. In §5, we also demonstrate how to incorporate existing policies in SiloD to make storage-aware joint allocations.

Caching algorithms and systems. Caching algorithms have been extensively studied in areas of processors, storage systems, content distribution networks, etc. These works share the same principle of leveraging information of access recency [37, 55, 56], frequency [14, 17, 26], or their combinations [12, 18, 23, 63] and using heuristics or machine learning techniques [19, 60, 73] to optimize cache hit ratio. General cluster caching systems like Alluxio [46] use classic caching algorithms like LRU and LFU as they do not have specific assumptions for workloads. Some systems, e.g., PacMAN[16] and Netco [39], leverage the characteristics of MapReduce-style jobs to improve storage efficiency, but they are specific to data analytic workloads and operate only on the storage level, independent from job scheduling. In comparison, SiloD exploits the access pattern of deep learning jobs to unify them by co-designing cluster scheduler and cache.

Data-aware schedulers. In the big-data era there also exist works that propose to make schedulers aware of data (e.g., data locality) [20, 28, 29, 38, 65, 66, 70]. For deep learning training, Hoard [58] leverages scheduling hints to prefetch dataset to local cache before jobs start, which is useful when there is redundant remote IO bandwidth thus orthogonal to SiloD. The key innovation of SiloD is making schedulers not only aware of but also able to control data storage, via the characteristics of deep learning, to improve global efficiency.

Multi-resource allocation. Multi-resource allocation has been extensively studied [27, 31, 61]. Synergy [49] also shows how to jointly consider GPU, CPU, and memory when scheduling deep learning training jobs. Different from these general multi-resource schedulers, SiloD observes that cache and remote IO are two interchangeable resource types: there are no predefined demands for the resources, and the allocation of one type can reduce the demand for the other. SiloD further builds a closed-form model for the two, thereby incorporating them in existing schedulers.

9 Conclusion

Motivated and facilitated by the unique characteristics of deep learning training, we advocate for the co-design of data caching and job scheduling. SiloD shows not only the feasibility but also the benefit of such a unified co-design, which treats cache and remote IO as first-class resources for joint allocation. With the insight to the unique data access pattern of deep learning training, we derive a simple but accurate closed-form performance estimator to calculate the performance impact of both computation and storage. Different state-of-the-art deep learning schedulers are integrated in SiloD via a unified framework, which shows great improvement on their respective scheduling objectives.

Acknowledgments

We thank the shepherd Ana Klimovic and the anonymous reviewers for their constructive feedback and suggestions. This work is partially supported by the National Key Research and Development Program of China (No. 2021ZD0110202).

References

[1] Creating a dataset and a challenge for deepfakes. https://ai.facebook.com/blog/deepfake-detection-challenge/, 2020.
[2] Open images dataset. https://opensource.google/projects/open-images-dataset, 2020.
[3] Amazon s3. https://aws.amazon.com/s3/, 2021.
[4] Amazon sagemaker. https://aws.amazon.com/sagemaker/, 2021.
[5] Aws gpu instances. https://go.aws/3DkWUgY, 2021.
[6] Azure blob storage. https://azure.microsoft.com/en-us/services/storage/blobs/, 2021.
[7] Azure data lake. https://azure.microsoft.com/en-us/solutions/data-lake/, 2021.
[8] Azure machine learning. https://ml.azure.com, 2021.
[9] Gpu optimized virtual machine sizes. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu, 2021.
[10] Azure storage scalability and performance targets for standard storage accounts. https://docs.microsoft.com/en-us/azure/storage/common/scalability-targets-standard-account, 2022.
[11] Nvidia gpu generations. https://www.nvidia.com/en-gb/data-center/products/, 2022.
[12] Marc Abrams, Charles R Standridge, Ghaleb Abdulla, Edward A Fox, and Stephen Williams. Removal policies in network caches for world-wide web documents. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 293–305, 1996.
[13] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675*, 2016.
[14] Charu Aggarwal, Joel L Wolf, and Philip S. Yu. Caching on the world wide web. *IEEE Transactions on Knowledge and data Engineering*, 11(1):94–107, 1999.
[15] Alex Aizman, Gavin Maltby, and Thomas Breuel. High performance i/o for large scale deep learning, 2020.
[16] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. Pacman: Coordinated memory caching for parallel jobs. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 267–280, 2012.
[17] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Performance Evaluation Review*, 27(4):3–11, 2000.
[18] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. {LHD}: Improving cache hit rate by maximizing hit density. In *15th {USENIX} Symposium*

on Networked Systems Design and Implementation ({NSDI} 18), pages 389–403, 2018.

[19] Daniel S Berger. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 134–140, 2018.

[20] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, 2014.

[21] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, 2016.

[22] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[23] Ludmila Cherkasova and Gianfranco Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *International Conference on High-Performance Computing and Networking*, pages 114–123. Springer, 2001.

[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[26] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–12, 2011.

[27] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[28] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. pages 365–378, 04 2013.

[29] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 99–115, 2016.

[30] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

[31] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

[32] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 689–706, 2022.

[33] Robbie Gruener, Owen Cheng, and Yevgeni Litvin. Introducing petastorm: Uber atg's data access library for deep learning. https://eng.uber.com/petastorm/, 2018.

[34] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[35] Guy Hacohen and Daphna Weinshall. On the power of curriculum learning in training deep networks. In *International Conference on Machine Learning*, pages 2535–2544. PMLR, 2019.

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

[37] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. {LAMA}: Optimized locality-aware memory allocation for key-value cache. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 57–69, 2015.

[38] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[39] Virajith Jalaparti, Chris Douglas, Mainak Ghosh, Ashvin Agrawal, Avrilia Floratou, Srikanth Kandula, Ishai Menache, Joseph Seffi Naor, and Sriram Rao. Netco: Cache and i/o management for analytics over disaggregated stores. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 186–198, 2018.

[40] Hervé Jégou, Florent Perronnin, Matthijs Douze, Jorge Sánchez, Patrick Pérez, and Cordelia Schmid. Aggregating local image descriptors into compact codes. *IEEE transactions on pattern analysis and machine intelligence*, 34(9):1704–1716, 2011.

[41] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Multi-tenant GPU clusters for deep learning workloads: Analysis and implications. *MSR-TR-2018-13*, May 2018.

[42] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[44] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 2020)*. USENIX, February 2020.

[45] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish your training data: Reusing partially augmented samples for faster deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 537–550, 2021.

[46] Haoyuan Li. *Alluxio: A virtual distributed file system*. PhD thesis, UC Berkeley, 2018.

[47] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens Van Der Maaten. Exploring the limits of weakly supervised pretraining. In *Proceedings of the European conference on computer vision (ECCV)*, pages 181–196, 2018.

[48] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient gpu cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[49] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond GPUs for DNN scheduling on Multi-Tenant clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, Carlsbad, CA, July 2022. USENIX Association.

[50] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *Proc. VLDB Endow.*, 14(5):771–784, jan 2021.

[51] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. Tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, jul 2021.

[52] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[53] NetApp. Deep learning and ai in the cloud with nfs storage. https://cloud.netapp.com/blog/ai-and-deep-learning-in-the-cloud, 2019.

[54] Ed Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*. USENIX, October 2012.

[55] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[56] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. Cflru: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241, 2006.

[57] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[58] Christian Pinto, Yiannis Gkoufas, Andrea Reale, Seetharami Seelam, and Steven Eliuk. Hoard: A distributed data caching system to accelerate deep learning training on the cloud, 2018.

[59] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, et al. Singularity: Planet-scale, preemptible, elastic scheduling of ai workloads. *arXiv preprint arXiv:2202.07848*, 2022.

[60] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, et al. Learning relaxed belady for content distribution network caching. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 529–544, 2020.

[61] Karnam Sreenu and M. Sreelatha. W-scheduler: Whale optimization for task scheduling in cloud computing. *Cluster Computing*, 22(1):1087–1098, jan 2019.

[62] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[63] Haisheng Tan, Shaofeng H-C Jiang, Zhenhua Han, Liuyan Liu, Kai Han, and Qinglin Zhao. Camul: Online caching on multiple caches with relaying and bypassing. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 244–252. IEEE, 2019.

[64] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[65] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[66] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J Franklin, and Ion Stoica. The power of choice in data-aware cluster scheduling. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 301–316, 2014.

[67] Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training. In *49th International Conference on Parallel Processing-ICPP*, pages 1–11, 2020.

[68] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[69] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020.

[70] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[71] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis CM Lau, Yuqi Wang, Yifan Xiong, et al. Hived: Sharing a gpu cluster for deep learning with guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 515–532, 2020.

[72] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1042–1057, New York, NY, USA, 2022. Association for Computing Machinery.

[73] Chen Zhong, M Cenk Gursoy, and Senem Velipasalar. A deep reinforcement learning-based framework for content caching. In *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, pages 1–6. IEEE, 2018.