Optimizing the Training of Co-Located Deep Learning Models Using Cache-Aware Staggering

Kevin Assogba[†], Bogdan Nicolae[λ], M. Mustafa Rafique[†]
[†]Rochester Institute of Technology, [λ]Argonne National Laboratory
[†]{kta7930, mrafique}@cs.rit.edu, [λ]bnicolae@anl.gov

***Abstract*—Despite significant advances, training deep learning models remains a time-consuming and resource-intensive task. One of the key challenges in this context is the ingestion of the training data, which involves non-trivial overheads: read the training data from a remote repository, apply augmentations and transformations, shuffle the training samples, and assemble them into mini-batches. Despite the introduction of abstractions such as data pipelines that aim to hide such overheads asynchronously, it is often the case that the data ingestion is slower than the training, causing a delay at each training iteration. This problem is further augmented when training multiple deep learning models simultaneously on powerful compute nodes that feature multiple GPUs. In this case, the training data is often reused across different training instances (e.g., in the case of multi-model or ensemble training) or even within the same training instance (e.g., data-parallel training). However, transparent caching solutions (e.g., OS-level POSIX caching) are not suitable to directly mitigate the competition between training instances that reuse the same training data. In this paper, we study the problem of how to minimize the makespan of running two training instances that reuse the same training data. The makespan is subject to a trade-off: if the training instances start at the same time, competition for I/O bandwidth slows down the data pipelines and increases the makespan. If one training instance is staggered, competition is reduced but the makespan increases. We aim to optimize this trade-off by proposing a performance model capable of predicting the makespan based on the staggering between the training instances, which can be used to find the optimal staggering that triggers just enough competition to make optimal use of transparent caching in order to minimize the makespan. Experiments with different combinations of learning models using the same training data demonstrate that (1) staggering is important to minimize the makespan; (2) our performance model is accurate and can predict the optimal staggering in advance based on calibration overhead.**

***Index Terms*—Deep Learning, Caching and Reuse of Training Data, Co-Located Training, Performance Modeling**

I. Introduction

Deep learning (DL) models are rapidly gaining traction both in the industry and scientific computing [1], driven by the accumulation of massive data and the computing capability of accelerators such as GPUs. In science, for example, instruments that collect data at GB/s and 100+ TB/day present a wide range of learning opportunities in areas such as fusion energy science [2], lattice quantum chromodynamics [3], ptychography [4], drug design and response prediction [5], etc.

Training a DL model is data-intensive and requires extensive computation, communication, and storage resources. For example, vision models such as VIT [6] and natural language models such as BERT [7] contain hundreds of millions of parameters. GPU accelerators and AI runtimes have evolved to take advantage of massive parallelism, enabling efficient training of such large DL models. However, the problem of feeding the training data to AI runtimes fast enough to take advantage of efficient training remains a key challenge.

Modern DL runtimes such as PyTorch [8] and TensorFlow [9] are beginning to acknowledge the importance of optimizing the entire training data life-cycle: from reading the training samples, augmenting them through transformations, shuffling them to simulate pseudo-random sampling, and finally grouping them together into mini-batches that are fed to the training. To this end, data pipeline abstractions were proposed (such as NVIDIA's DALI [10]) that asynchronously overlap the steps in the training data life-cycle with actual training steps. However, despite such asynchronous overlaps, the data pipeline may not keep up with the training steps, in which case each training step needs to wait until the next mini-batch is available [11]. This may happen especially when the training data is stored on a remote repository (such as a parallel file system), in which case high I/O latencies and insufficient I/O bandwidth are to blame for stalls in the data pipeline [12]. In such scenarios, up to 85% of the training time may be spent waiting for the data pipelines [13], [14].

This issue is amplified by the fact that modern HPC systems feature compute nodes equipped with several GPUs that compete for the limited I/O bandwidth. Specifically, each GPU typically runs a different DL model training instance that is attached to a different data pipeline, which means the data pipelines compete for the I/O bandwidth to read the training samples from the repository. Fortunately, in a large number of scenarios, the training instances are related and share the same training data. For example, this is the case for multi-model learning (i.e., train different DL models to solve the same problem, and use them in tandem to increase the confidence in the inference results), neural architecture search (automated exploration of DL model candidates feasible to solve a problem), hyper-parameter optimization (fine-tuning of DL model parameters such as learning rate and dropout). Furthermore, it is often the case that different users (that are unaware of each other) make use of the same standardized training data (e.g., ImageNet [15]) to train their DL models.

Caching the training data is one possible solution to mitigate the high I/O overheads of data pipelines [16]. It may even happen automatically. For example, modern operating systems use the spare memory available on the compute nodes to cache the data read from POSIX file systems. Under ideal circumstances, after training for an epoch (during which the

full training data is visited exactly once), subsequent epochs (and other training instances sharing the same training data) would be able to benefit from local caching to reduce or even eliminate remote I/O to external repositories.

However, as the cost of traditional training that involves a large number of epochs is increasing, techniques such as fine-tuning a DL model using transfer learning [17] or applying an early stopping policy during neural architecture search [18] combined with transfer learning [19] is becoming more popular. In this context, DL training is *short-lived* and runs for one or a few epochs only. Furthermore, it is often the case that they need to be scheduled *simultaneously* (e.g., because they are part of an ensemble). Under such circumstances, the interleaving of different access patterns causes high cache contention, misses, and trashing, which limits the reuse opportunity of cached training data and therefore the overall effectiveness of caching.

In this paper, we focus on the problem of how to efficiently train co-located DL models that share the same input data stored initially on a remote repository. For simplicity, we focus on the case of pairs of DL models. In this case, our goal is to *minimize the training makespan*, i.e., the duration until both DL models are fully trained. As a secondary goal, we aim to simultaneously *reduce the resource utilization* needed for the training. We assume that the scheduler has the freedom to start the two DL model training instances in any configuration: simultaneously, serially (one after another), or staggered (start one of the training instances after a delay). In either case, we define resource utilization as the sum of the runtimes of the individual training instances on the GPUs. In other words, if we have the choice between starting two training instances simultaneously or staggered, and in both cases, the makespan is the same, we prefer the staggered configuration, because in this case, we can assign other work on the GPU that is scheduled to run the staggered training instance later.

A key observation that we leverage is the fact that it is possible to use the same pseudo-random number generator seed in the data pipelines of both DL models we aim to train simultaneously. Using this approach, the training data will be read in the same order by both training instances, thereby negating the cache-trashing effect, since we avoid the interleaving of different access patterns. On the other hand, the contention for I/O bandwidth is more difficult to address, as it is subject to a trade-off: if we start both training instances at the same time, then this allows the maximum degree of training parallelism at the cost of high I/O cache contention. At the other extreme, if we run the training instances serially, we avoid I/O contention at the cost of no training parallelism. To solve this trade-off, we have to find the *optimal staggering*, which measures how long to wait after starting the first training instance until we start the second training instance, such as to minimize the training makespan. To this end, we introduce several contributions, summarized below:

- We study the impact of I/O overheads during the training of both individual and pairs of DL models and identify key metrics that enable the characterization and mitigation of caching and I/O contention triggered by the data pipelines (Section IV).
- We introduce a performance model that requires minimal calibration in the form of training the pair of DL models for a few training steps under well-defined circumstances derived from the metrics used in the characterization. The model aims to predict the optimal staggering that minimizes the training makespan, which indirectly also reduces resource utilization. To this end, the performance model leverages piece-wise consistent behavior patterns (thanks to the iterative nature of DL model training), which are composed as a state machine (Section V).
- We demonstrate the effectiveness of the performance model to minimize the makespan and reduce the resource utilization for the training of different pairs of popular DL models and standardized datasets used in the AI community. To this end, we run extensive experiments to measure the training makespan of all possible staggerings and show that our model can predict the optimal staggering with an error of less than 1%, while reducing the training makespan and resource utilization by more than 50% compared with several baseline approaches (Section VI).

II. Related Work

Deep Learning I/O Optimization: Data movement is a key performance bottleneck in modern DL training applications [20], [12] as approximately 62% of machine learning workloads observe at least 1 ms of wait time and 16% spend at least 100 ms waiting for the input data [21]. Stalls are introduced by over-the-network data transfer, slow data pipeline transformations, large discrepancies between modern hardware accelerators and CPU processing speeds (especially when transformations are performed on CPUs to avoid contention for GPUs, where the training is performed) [22]. In particular, the I/O access patterns (small, pseudo-random accesses) are particularly challenging for traditional POSIX-based storage systems such as parallel file systems (PFS) [23]. Pumma et al. [24] have shown that memory-mapped databases can be adapted to reorganize the training data such as to maximize OS-level caching benefits and minimize POSIX metadata overheads (e.g., enumerate files in a directory). Other approaches such as FanStore [25] provide a global cache layer on node-local burst buffers in a compressed format, allowing POSIX-compliant file access to the compressed data in user space. Data pipelines that offer a streaming view over the training data are becoming the norm in AI training, with industry-standard approaches such as NVIDIA DALI [10] offering asynchronous implementations that hide most overheads by overlapping I/O reads, transformations, shuffling and batching with the training. Such approaches are insufficient at a large scale, as many training instances may share the training data and therefore compete for the limited I/O bandwidth of the repository storing it. In this case, collaborative caching of the training data on the compute nodes is a popular technique. Specifically, the compute nodes access each other's cache preferentially, which reduces the I/O contention on the repository.

This approach can be further optimized by maximizing the reuse of training data thanks to the foreknowledge of the access pattern [14], [13]. Other complementary directions are the use of low-latency emerging memory technologies, e.g., CXL [26], or specialized DL model repositories to store and/or cache candidates viable for transfer learning [27], which is a scenario particularly relevant for short-lived training instances.

Performance Prediction: A large number of performance modeling solutions rely on observations from traces of historical application executions to predict the future application performance [28], [29]. One possible approach is to use ML-based numerical models to identify representative patterns from collected data [30]. When the application generates specific repetitive behavior patterns of performance metrics over time (both in terms of runtimes or CPU, memory, and network utilization), sequence-to-sequence DL models (originally applied in natural language processing) have been successfully adapted to identify these behavior patterns in an online fashion [31]. Analytical approaches [32], [33] are also used to collectively or separately model the computation, communication, and I/O performance of DL workloads. They are based on the fundamental observation that DL workload executions follow a repetitive pattern where one mini-batch of data is processed per iteration and the entire dataset is read in each epoch [34]. Furthermore, learning runtime properties by sampling tasks may also be used to predict properties of the whole workload and avoid the assumption of cyclic execution patterns [35]. Combining both numerical and analytical solutions further optimizes the prediction process as fewer data points need to be collected. It is important to note that while such approaches can be effective at predicting the behavior of a single DL training instance, in our scenario we are interested in multiple DL training instances that are co-located and compete for resources, which makes the problem more challenging.

Concurrent Execution of Co-Located DL Workloads: The concurrent execution of co-located DL workloads leads to workload interference through resource contention, bandwidth bottleneck, race conditions, etc [36]. Different workloads can be scheduled on dedicated GPUs to provide isolation to training processes. However, this configuration does not eliminate interference due to data pipelines competing for I/O and CPU resources. In this regard, one particular aspect that has proved successful is the load balancing of the stages of the data pipeline across the CPU cores [13] (i.e., allocate CPU cores to data pipeline stages proportionally to their computational complexity). Nevertheless, the competition for I/O resources and OS-level caches leads to unpredictability in workload performance depending on the co-located workload and resource availability [37]. As a consequence, efforts have emerged to optimize the training data layout and caching strategies for frequent reuse [38], [39]. For example, Meta has built a central data warehouse for training instances that heavily filter massive and evolving datasets before reusing them. Nevertheless, such efforts are often targeted at domain-specific areas (e.g., recommender systems [40]).

To our best knowledge, we are the first to study the benefits of optimal staggering of co-located DL model training instances for the purpose of mitigating the contention of the data pipelines through shared caches of training data, which ultimately reduces the makespan and resource utilization.

III. Background and Problem Formulation

Data pipelines abstract input data as a potentially infinite sequence of training samples, e.g., tensors or composite types (tuples, nested datasets, etc.). Training samples are not accessed individually, but in groups called mini-batches that are assembled into a *batch queue*, working like an iterator used to feed a new mini-batch at each training iteration. The path from reading the input data to generating the mini-batches creates a complex multi-stage producer-consumer pipeline, as illustrated in Figure 1. Specifically, the training samples are read from the repository (typically as files stored on a PFS), encoded as tensors, optionally transformed using custom augmentation functions, shuffled, and finally assembled into mini-batches, which are finally enqueued into the batch queue.

A typical implementation of a data pipeline (such as NVIDIA's DALI [10]) is *asynchronous*, i.e., it fills the batch queue in the background, without blocking the training iterations. If the data pipeline cannot keep up with the training iterations (i.e., the training iterations consume mini-batches from the batch queue faster than the data pipeline can produce them), the training needs to wait for the data pipeline between the iterations. Such I/O wait delays increase the overall duration of the training. An example is illustrated in Figure 2: the data pipeline assembles mini-batches one and two quickly, but takes much longer for mini-batch three. Although a large part of the overhead of assembling mini-batch three is overlapped with the second training iteration, this still causes a significant I/O wait delay until the third training iteration can start. Furthermore, while these data pipelines are highly optimized and can take advantage of both GPUs and CPUs to parallelize the intermediate stages (decoding, transformations, shuffling, etc.), I/O requests often become a weak link in the pipeline, especially when they need to be served by a PFS, bottlenecking the rest of the stages [12], [13].

For the purpose of this work, we assume two DL training instances A and B that share the same training data. To take advantage of caching, they are co-located on the same compute node but run on different GPUs. If A and B overlap during their runtime, then they will either compete for the I/O bandwidth to the remote repository (if they need to access different training samples that are not cached locally) or they will compete for the cache (both for reads and writes). Our goal is to minimize the makespan of finishing the training of both A and B while reducing the GPU resource utilization necessary to achieve this objective.

Since both A and B visit all training samples of the same dataset exactly once during an epoch, a naive strategy could simply start both A and B at the same time and let them compete for I/O bandwidth to the remote repository, under the assumption that any first-time read of a training sample can be cached locally, therefore the I/O overhead of accessing

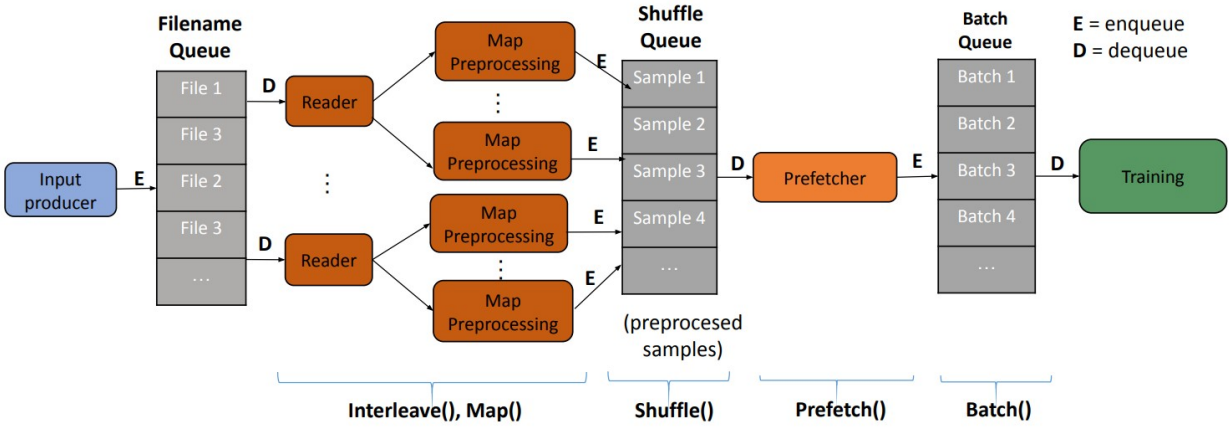

Fig. 1. Data pipeline: multi-stage streaming of training data from a remote repository.

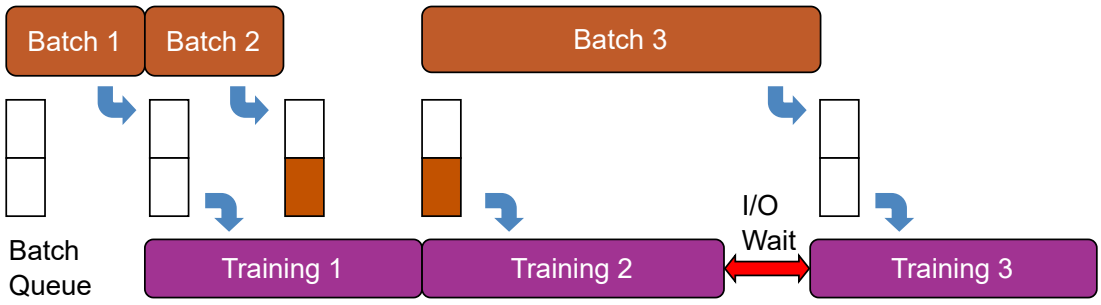

Fig. 2. Data pipeline: mini-batches that accumulate slowly in the batch queue cause I/O wait delays that slow down the training.

the remote repository is paid only once, regardless of which training instances issued the first time read. We explain why such a naive strategy is sub-optimal in the next section.

IV. Study of I/O Bottlenecks during Co-Located Training with Variable Staggering

In this section, we aim to characterize the I/O and caching behavior of co-located DL training instances that share the same training data. It simultaneously motivates our contribution and explains key behavior patterns and metrics that are leveraged by our contribution.

Since the I/O bandwidth to the remote repository is shared by the two co-located DL training instances A and B, a naive strategy that simply starts them at the same time amplifies the I/O wait delays due to competition for limited I/O bandwidth, especially when A and B visit the training data in a different order. A simple fix to solve this issue would be to force A and B to visit the training data in the same order, which would maximize the reuse of locally cached training data.

We propose to achieve this by fixing the pseudo-random number generator seed used by the data pipelines of A and B to sample the training data, which effectively results in a deterministic order of visiting the training data. However, a key question is whether such a simple fix is enough to enable the naive strategy to achieve our goal of minimizing the makespan.

As expected, this is not the case. To illustrate this point, we construct an experimental setup that concurrently trains two DL models (A: *ResNet-50* and B: *EfficientNet-B0*), commonly used as benchmarks, on two GPUs of the same compute node. These DL models use the same standardized training data (TinyImageNet [41]) available on a remote repository (GlusterFS) and begin with a cold cache. For completeness, please refer to the full description of the setup in Section VI-A.

We fix the pseudo-random number generator seed for the data pipelines of A and B to implement a deterministic read order of the training samples for the naive strategy. We say the naive strategy has a staggering of 0% because both A and B start at the same time. Then, we compare with alternative experiments that start *ResNet-50* first, then wait until X% of the total runtime of *ResNet-50* has passed (100% denotes the total runtime of *ResNet-50* when running standalone, without competition for I/O bandwidth), then start *EfficientNet-B0*. In this case, we say the staggering is X%. A staggering of 100% corresponds to the case when A and B run serially.

We depict the results in Figure 3. Indeed, as can be observed, the naive strategy does not produce the minimum makespan and leads to high resource utilization. By using a staggering of 40%, the makespan can be reduced, while at the same time the resource utilization of *EfficientNet-B0* is 40% lower since it starts later but finishes faster. In general, we note the following important observations:

Observation 1 - I/O competition negatively impacts the individual runtimes and resource utilization of both training instances: Even with a fixed visiting order of the

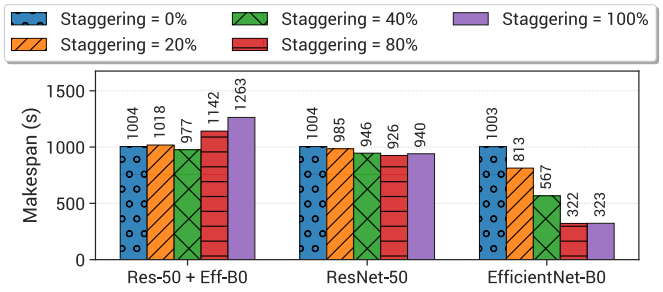

Fig. 3. Impact of variable staggering (delay of starting EfficientNet-B0 relative to ResNet50) on the training makespan. Lower is better.

training data, the competition for the I/O bandwidth of the remote repository and the local cache creates a significant bottleneck that slows down both DL training instances.

Observation 2 - The staggered training instance experiences significant reduction in individual runtime and resource utilization due to caching: Specifically, *EfficientNet-B0* is almost 3x faster when most of its training data was already cached by *ResNet-50*, which demonstrates that caching effectively mitigates the I/O overheads of accessing the remote repository that stores the training samples, as long as there is no high I/O contention between the remote reads that populate the cache and the cached reads of the staggered instance.

Observation 3 - Staggering reduces the makespan up to an optimal point, then it experiences diminishing returns: Since our primary goal is to minimize the makespan, not the runtime and resource utilization of *EfficientNet-B0* from a selfish perspective, we cannot simply wait until the training data is fully cached by *ResNet-50*. In this case, there is a trade-off: the training makespan of *ResNet-50* and *EfficientNet-B0* (Res-50 + Eff-B0) decreases as the staggering increases up to 40%, then it begins increasing again, which confirms that reducing I/O competition helps up to a point, after which it negatively impacts the makespan.

Observation 4 - Minimizing the makespan through staggering simultaneously reduces resource utilization: Not only does the optimal staggering reduce the makespan (even if by a small degree), but at the same time, it significantly reduces resource utilization because the staggered training instance does not consume any resources before it starts. Therefore for the duration of the staggering, GPU, CPU, or I/O resources can be used for other purposes.

To explain these observations better, in Figure 4 we depict the I/O wait time during the first epoch of DL training for a variable degree of staggering. The I/O wait time measures the time interval between the beginning of the training iteration and the moment when a mini-batch was successfully dequeued from the batch queue of the data pipeline. Using the I/O wait time, we can study how well the data pipeline hides the I/O overheads asynchronously from the training steps.

As can be observed, starting from a cold cache results in a constantly high I/O overhead in the case of *ResNet-50*. However, as the staggering of *EfficientNet-B0* increases, so does the effectiveness of the OS-level caching at reducing the I/O wait time of *EfficientNet-B0*. Eventually, *EfficientNet-B0* catches up with the cache of *ResNet-50*, at which point both training instances begin to experience high I/O wait times. Thus, we propose that leveraging I/O wait times under different circumstances (cold vs. warm cache, standalone vs. concurrent execution subject to I/O competition) is an important step towards a performance model that can predict the optimal staggering.

V. Optimal Cache-Aware Staggering using Performance Modeling

In this section, we present our key contribution: based on the observations detailed in Section IV, we propose a performance model and methodology to find the optimal cache-aware staggering that simultaneously minimizes the training makespan and resource utilization of co-located training instances that share the same input data.

A. Performance Model

To this end, we introduce the following notations: the first training instance A starts at timestamp $t_A = 0$, exhibits an average duration I_A of the training iterations which is extended by an average I/O wait time W_A. Furthermore, we assume the total number of training iterations is fixed per epoch (full pass over the training data). A trains for a total of E_A epochs. Similarly, training instance B starts at timestamp $t_B \geq 0$, has an average training iteration duration I_B and an average I/O wait time W_B. It trains for E_B epochs.

We force the data pipelines of both training instances to use the same seed for their pseudo-random number generators used for shuffling the training data. Thus, if $t_B > 0$, then B will initially consume the cached training samples of A (instead of accessing them directly from the remote repository). Assuming the I/O wait time of B for accessing the warm cache is $W_{B|cached}$, if $I_B + W_{B|cached} < I_A + W_A$, then B will eventually "catch up" with the cache of A, i.e., there exists t_C such that A and B have processed the same number of mini-batches. We denote $N_A(t)$ and $N_B(t)$ as the number of mini-batches (training iterations) processed at moment t by each of A and B respectively. Thus, $N_A(t_C) = N_B(t_C) = N$. Then, $(W_A + I_A) \times N = t_C$ and $(W_{B|cached} + I_B) \times N = t_C - t_B$. By solving these equations, we obtain:

$$t_C = \frac{t_B}{1 - \frac{W_{B|cached}+I_B}{W_A+I_A}} \qquad N = \frac{t_C}{W_A + I_A} \tag{1}$$

Then, if we denote the runtime of the first epoch of A as T_{A1}, the runtime of the first epoch of B as T_{B1}, the I/O wait time of A under concurrency $W_{A|conc}$, the I/O wait time of B under concurrency $W_{B|conc}$, and the total number of mini-batches in an epoch as N_E, we have:

$$\begin{aligned} T_{A1} &= t_C + (N_E - N)(W_{A|conc} + I_A) \\ T_{B1} &= t_C - t_B + (N_E - N)(W_{B|conc} + I_B) \end{aligned} \tag{2}$$

Accordingly, the total runtimes T_A and T_B of training instances A and B over E_A and E_B epochs are:

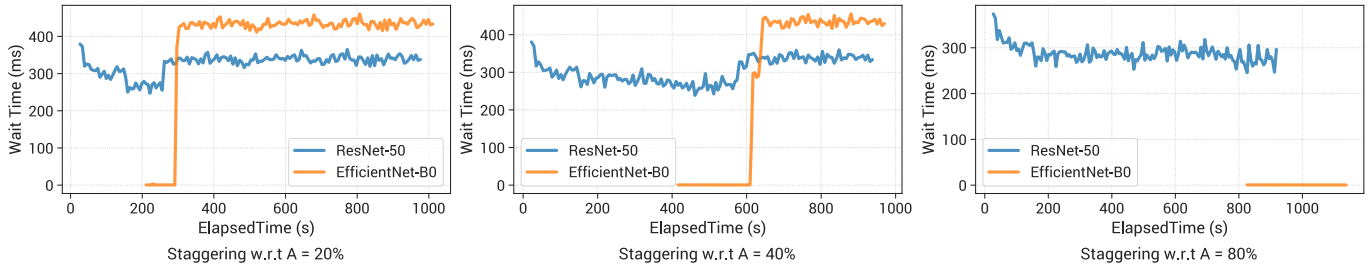

Fig. 4. Analysis of the I/O wait time during the first epoch of DL training for a variable staggering.

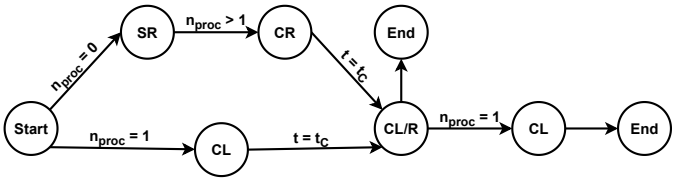

Fig. 5. Representation of performance mode states. SR: Standalone, Remote Data; CR: Concurrent, Remote Data; CL: Concurrent, Local (cache) Data; CL/R: Concurrent, Local (cache) or Remote Data.

$$T_A = T_{A1} + \sum_{e=2}^{E_A} N_E \times \left(W_{A|cached} + I_A\right)$$
$$T_B = T_{B1} + \sum_{e=2}^{E_B} N_E \times \left(W_{B|cached} + I_B\right) \quad (3)$$

Finally, we obtain the makespan $M = max\left(T_A, T_B + t_B\right)$. Thus, the optimal staggering denoted OS is:

$$OS = \arg\min_{t_B}[max\left(T_A, T_B + t_B\right)] \quad (4)$$

In case $I_B + W_{B|cached} > I_A + W_A$, then B will not "catch up" with the cache of A, therefore, it will always benefit from caching and the optimal makespan is easy to calculate.

To account for all these alternative scenarios, we introduce a state machine (depicted in Figure 5) that defines the piece-wise behavior of the performance model, which simplifies the implementation of our performance model. The state machine accounts for the different I/O behaviors of the two training instances during I/O competition as their runtime overlaps.

B. Predicting the Optimal Staggering

Using the performance model described above, we predict the optimal staggering using two steps:

1) Obtain I_A, I_B, W_A, W_B, $W_{A|cached}$, $W_{B|cached}$, $W_{A|conc}$, $W_{B|conc}$ by means of micro-benchmarking in order to calibrate the performance model.
2) Apply binary search to find the optimal staggering by predicting the training makespan at each halving step based on the performance model.

Note that the duration of the training iterations and the I/O wait delay is stable, both when running the training instances standalone and under concurrency, as discussed in Section IV. Therefore, it is enough to run only a small number of training iterations in order to calibrate the performance model. Furthermore, these parameters can be archived and reused later if the same DL models are trained again, which is often the case when the training data or the hyperparameters change for an ensemble of DL models that train together.

TABLE I
SUMMARY OF THE PARAMETERS DESCRIBING THE DL MODEL PAIRS EVALUATED IN OUR EXPERIMENTS.

Set	Pairs of models	Parameters (Millions)	Batch Size	Number of iterations	I/O duration per iteration (ms)	Training duration per iteration (ms)
1	A: Inception-V3	23.9	64	1560	489.9	258.7
	B: ResNet-50	25.6	64	1560	449.3	303.6
2	A: ResNet-50	25.6	64	1560	449.3	303.6
	B: EfficientNet-B0	5.3	64	1560	543.3	204.7
3	A: VGG-16	138.4	64	1560	278.8	462.2
	B: Inception-V3	23.9	64	1560	489.9	258.7

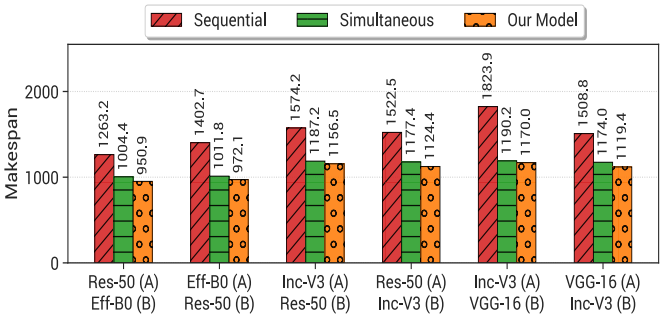

Fig. 6. Training makespan for the compared approaches. Notations: Res-50 (ResNet-50), Eff-B0 (EffientNet-B0), Inc-V3 (Inception-V3). Lower is better.

Another important aspect to note is that our approach is flexible and can predict the optimal staggering even after a DL training instance has already started. In this case, it is enough to simply perform the binary search between $X\%$ and 100%, with $X\%$ denoting the current progress of the DL training instance. By predicting the optimal staggering, makespan and resource utilization with our approach, schedulers can explore alternatives to decide how to co-locate pairs of DL training instances that share the same training data. For example, it may be possible that multiple training instances with different rates of progress are a possible candidate for co-location, but without anticipating the benefits of pairing a new training instance with each of them, the decision would be arbitrary and would cause sub-optimal makespan and/or resource utilization.

C. Implementation Details

We implemented our approach for PyTorch 1.13, using the data pipeline provided by NVIDIA DALI 11.0. Note that DALI is designed to be compatible with a variety of AI runtimes. Therefore, our approach can be trivially extended to support such AI runtime alternatives, including TensorFlow.

Specifically, we instrumented DALI's batch queue to measure and log the I/O wait time of the data pipeline at fine granularity for each iteration. Using the instrumented data pipeline, we designed and developed a Python framework to automate the calibration of the performance model. In this regard, we run each DL model training for a small number K of iterations in various combinations: standalone with a cold cache that forces remote reads from the repository (to obtain I_A, I_B, W_A, W_B), standalone with pre-cached training data (to obtain $W_{A|cached}$, $W_{B|cached}$), concurrent with cold cache (to obtain $W_{A|conc}$, $W_{B|conc}$). The final values of the parameters are calculated as the averages of the K iterations and stored in a catalog for future reuse (in case the same DL models are trained again with different training data or hyperparameters). Note that the reuse of the performance model parameters is partially possible even when only one of the DL models of the pair was trained before. For example, if A was trained before together with B, but now needs to be trained together with C, then we can directly reuse I_A, W_A, $W_{A|cached}$ and we only need to perform micro-benchmarks to obtain I_C, W_C, $W_{C|cached}$, $W_{A|conc}$, $W_{C|conc}$. By reusing these parameters, we eliminate the overhead of performing the calibration.

Finally, we implemented the state machine describing the piece-wise behavior of the performance model (introduced in Section V-A) and the binary-search algorithm (introduced in Section V-B) as a Python script that returns the optimal staggering. It transparently interacts with the catalog and performs the minimum amount of micro-benchmarking to obtain any missing performance model parameters. Our approach is generally applicable to any type of DL model as we do not use parameters specific to the neural architecture. Instead, our approach takes advantage of the iterative nature of the training process and specifically the constant training time per mini-batch during an epoch, which is true for the large majority of training methods.

VI. Evaluation

A. Experimental Setup

We evaluate our performance prediction model on the Chameleon Cloud testbed [42], a cloud research environment for experimentation. Our setup consists of two nodes, each equipped with 24 hardware cores of Intel Xeon E5-2670 CPUs (48 threads), 1 TB SATA hardware drive, 125 GB DDR4 memory, and 2 NVIDIA P100 GPUs. Each GPU has 16 GB of memory. Compute nodes are also attached to a 2 TB GlusterFS PFS located in the same data center and accessible through a POSIX mount point. The storage node serves all training datasets with the GlusterFS 9.6 server, and compute nodes are equipped with a GlusterFS 9.6 client. All experiments are executed with a software stack that consists of CUDA 11.2, PyTorch 1.13, and NVIDIA DALI 11.0.

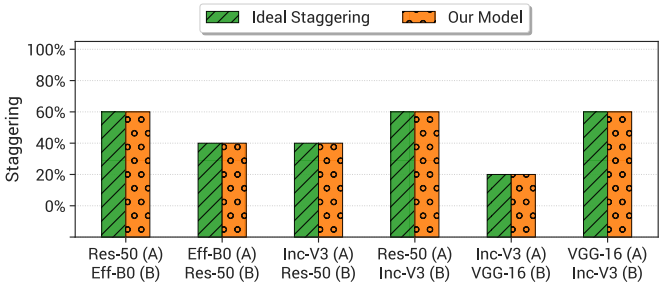

Fig. 7. Accuracy of our approach compared with the ideal staggering. Notations are identical as in Figure 6.

B. Compared Approaches

Throughout our evaluations, we use the following baselines for comparison:

1) **Sequential**: We run the DL training instances sequentially, starting with a cold cache: first A is allowed to run to completion, after which B is started and fully benefits from a warm cache. The training makespan is the sum of the runtimes of A and B.
2) **Simultaneous**: We run the DL training instances simultaneously, starting with a cold cache. In this case, both A and B compete for the I/O bandwidth to the GlusterFS repository. The training makespan is the duration of the slowest among A and B.
3) **Ideal Staggering**: This is an exhaustive search that tries all possible staggering configurations in increments of 10% relative to the runtime of A to experimentally determine the ideal staggering for B. It is intended as a theoretical baseline that enables us to evaluate the accuracy of the predictions of our performance model.
4) **Our Approach**: This is our approach as detailed in Section V. The calibration needed to obtain the parameters of the performance model was run for 100 iterations of the first epoch and the results were averaged.

C. Summary of the workloads

We evaluate three pairs of co-located DL training instances. For each pair, we study both the case when A starts first and B follows, as well as the opposite. The three pairs were obtained using four different standardized DL models frequently used in the AI community: EfficientNet-B0, Inception-V3, ResNet-50, and VGG-16. The training data used by each DL model is the TinyImagenet [41] dataset, which consists of a subset of images from the larger ImageNet [15] dataset. TinyImagenet is also frequently used in the AI community. The characteristics of the DL models and pairs are summarized in Table I.

D. Results: Training Makespan

We first analyze the training makespan for the six different configurations corresponding to the pairs summarized in Table I. The results are reported in Figure 6. As can be observed, our approach obtains the lowest training makespan compared

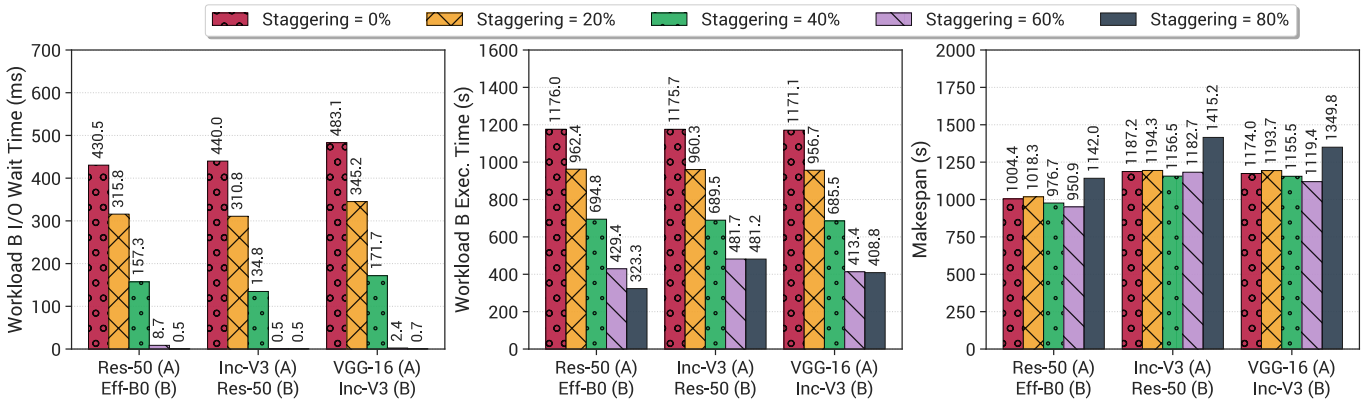

Fig. 8. Impact of staggering on I/O wait, execution time of B (second training instance), and makespan. Lower is better.

with both the sequential and the simultaneous approaches, for all configurations. As expected, the sequential approach is the slowest, followed by the simultaneous approach and our solution. Interesting to observe that the sequential approach is only 25% slower than the simultaneous approach, which shows that the I/O overheads are large and introduce long I/O wait times, thereby severely degrading the training performance.

Compared with the simultaneous approach, our approach reduces the training makespan by a small amount (up to 5%). However, as noted in Section IV, this small benefit is complemented by much lower resource utilization due to the execution of instance B at a later time, which we will discuss in the following sections. The fact that we can reduce the makespan compared with the simultaneous approach in all configurations is also significant in that it highlights the recurring bottleneck caused by competing for the I/O bandwidth to the remote repository and the local caches, despite different degrees of asynchronous overlapping between the data pipelines and the training iterations. We believe this finding will be essential in the design of next-generation data pipelines for multi-tenant DL training.

E. Results: Accuracy of the Optimal Staggering Prediction

Next, we focus on the accuracy of the optimal staggering predictions using our approach, which is depicted in Figure 7. As can be observed, our approach predicts the optimal staggering within an error of less than 1% compared with the ideal staggering, which is determined using an exhaustive search that experimentally evaluates the makespan for all possible staggering values (in increments of 10% relative to the total standalone runtime of the first DL model training instance when starting with a cold cache). This demonstrates that our proposal is accurate despite using little information obtained through a low-overhead calibration involving only a few training iterations.

Furthermore, another important observation is that the optimal staggering is mostly between 40%-60% of the runtime of the first training instance, which means that we can afford to wait for a long time before starting the second DL training instance, thereby dramatically reducing its runtime and therefore resource utilization (50% on average).

F. Results: Zoom on Different Levels of Staggering

Next, we study the impact of different degrees of staggering (ranging from 0% to 80%) on the average I/O wait delay, execution time of the second training instance B, and the training makespan. This study helps us explain the findings we presented in the previous sections. To this end, we rely on the measurements obtained from the experiment that determines the ideal staggering using an exhaustive search, as described in Section VI-E. The results are illustrated in Figure 8.

Starting with the I/O wait delay, we can observe high values for a staggering of 0% for the considered pairs. This means the second training instance experiences minimal benefits from caching due to the I/O competition. As the stagger increases, the average I/O wait delay decreases, indicating that the second training instance quickly consumes the cached training samples until it "catches up" with the first training instance, then it experiences I/O bottlenecks again. Interesting to note is that the I/O wait delay becomes negligible after a staggering of 60%, which correlates well with our previous findings that identified the optimal staggering of 60% for pairs.

As expected, the average I/O wait delay is a determining factor in the runtime of training instance B, since these delays are accumulated at every training iteration. Specifically, these I/O wait delays are large and can cause up to 3x slower runtime under competition for I/O bandwidth in the case of simultaneous start (staggering 0%).

An interesting trend is observable regarding the minimum makespan: the optimal staggering is not necessarily correlated with the minimum runtime of training instance B. For example, *EfficientNet-B0* achieves the minimum runtime when staggered 80% with respect to *ResNet-50*, which is visibly lower than the runtime at a staggering of 60%. However, the optimal makespan staggering is at 60%, not at 80%. Same observation applies for *Inception-V3* and *ResNet-50*. This confirms the observations in Section IV: staggering makes better use of caching and reduces the makespan up until a point, after which it provides diminishing returns.

Nevertheless, even if our main goal is to reduce the training makespan, the optimal staggering is always close to the minimum runtime of training instance B. Thus, our approach

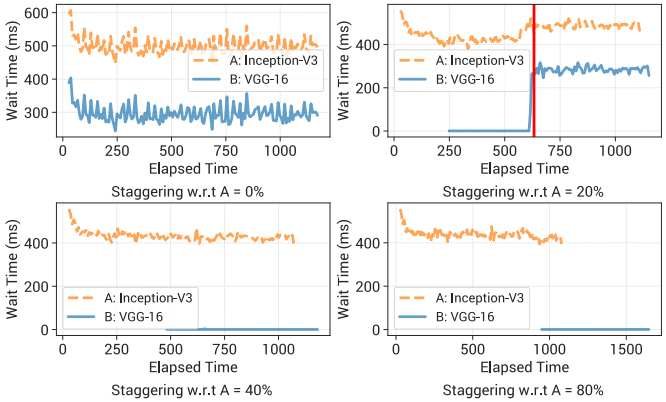

Fig. 9. I/O wait time for Inception-V3 and VGG-16 with a variable staggering (ranging 0%-80%).

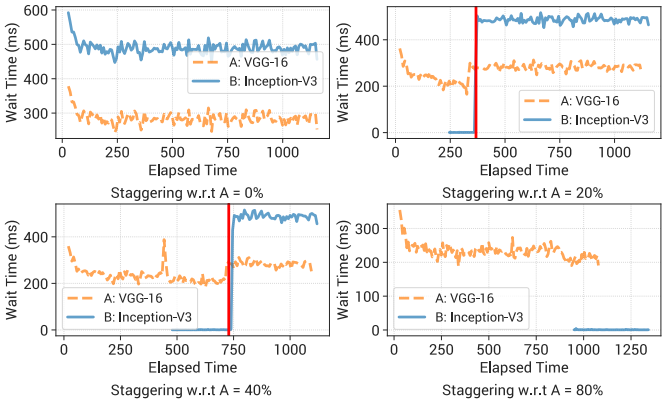

Fig. 10. I/O wait time of VGG-16 and Inception-V3 with a variable staggering (ranging 0%-80%).

achieves its second goal of enabling a significant reduction in resource utilization, since a large delay in the start of training instance B allows the GPU to be used for other workloads.

G. Results: Zoom on the Evolution of I/O Wait Time

Our final set of results focuses on explaining the average I/O wait delays discussed in the previous section. Figure 9 and Figure 10 zoom on the evolution of the I/O wait delay during the runtime of the VGG-16 and Inception-V3 DL model pair. The moment when the second training instance B fully consumes the training data cached by A (denoted t_C in Section V) is marked using a horizontal bar.

For example, we notice when the staggering is 20%, VGG-16 (in the role of A) loads data directly from the cache for more than 250 seconds, whereas Inception-V3 (in the role of B) consumes all cached data within 150 seconds. Among the two training instances, VGG-16 has a longer training time per mini-batch. Therefore, at the start of B, A has only cached 460 mini-batches of the dataset. In the reverse scenario where Inception-V3 is in the role of A and VGG-16 is in the role of B, 800 mini-batches are cached when VGG-16 starts. This gives VGG-16, in the reverse scenario, almost double the amount of cached training data to amortize the I/O wait delays until it catches up. Furthermore, since VGG-16 has slower training iterations, it will consume this larger amount of cached training data slower, giving more time to Inception-V3 to further populate the cache meanwhile.

Thus, it is important to carefully consider the order of execution of the DL workloads to minimize both the training makespan and resource utilization.

VII. Conclusion and Future Work

In this paper, we focus on the problem of how to efficiently train co-located DL models that share the same input data stored initially on a remote repository such as a parallel file system. Specifically, given a pair of DL model training instances, our goal is to *minimize the training makespan*, i.e., the duration until both DL models are fully trained. As a secondary goal, we aim to simultaneously *reduce the resource utilization* needed for the training.

Based on experimental evaluations, we have observed that simply fixing the pseudo-random number generator of the asynchronous data pipelines employed by AI runtimes in order to obtain a deterministic read order of the training samples for both instances is not enough to simply employ a naive strategy that starts both training instances simultaneously and takes advantage of local caching on the compute nodes. The main reason for this observation is the high competition for I/O bandwidth and local caches, which introduces large I/O wait delays in the data pipeline and therefore increases the training duration. Instead, a better strategy is to stagger one of the training instances, which reduces the penalty of competition and improves both the training makespan and resource utilization (i.e., the staggered training instance starts later and runs for a shorter runtime, thereby enabling GPUs to be used for other workloads).

To this end, we proposed a performance model and methodology to find the optimal staggering that produces the minimum makespan, which also reduces resource utilization. Our approach is flexible and can be applied to take initial decisions (i.e., how to schedule two related training instances that share the same training data) as well as dynamic decisions (i.e., anticipate the benefits of co-locating a training instance with another that already started). Compared with several baselines, our approach obtains a shorter makespan, while at the same time reducing the resource utilization of the staggered training instances by more than 50%.

Encouraged by these results, we plan to extend our work in the future in several directions: (1) model the performance of multiple DL training instances that are dynamically submitted by capturing the probability distribution followed by the workloads submission or considering the makespan of two instances as one and optimize the third instance accordingly; (2) investigate how these findings can be used to design and develop novel data pipelines for DL training that are aware of co-located DL training instances that share the same training data and provide optimized cache-sharing strategies under concurrency for this purpose; (3) integrate our performance model into an HPC cluster scheduler to improve the orchestration of DL training workloads.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Advanced Scientific Computing Research, under under contract DE-AC02-06CH11357. Furthermore, it was supported by the National Science Foundation (NSF) under Awards No. 2106634 and 2106635.

REFERENCES

[1] X. Wu, V. Taylor, J. M. Wozniak, R. Stevens, T. Brettin, and F. Xia, "Performance, energy, and scalability analysis and improvement of parallel cancer deep learning candle benchmarks," in *Proc. ICPP*, 2019, pp. 1–11.

[2] I. Joseph, Y. Shi, M. Porter, A. Castelli, V. Geyko, F. Graziani, S. Libby, and J. DuBois, "Quantum computing for fusion energy science applications," *Physics of Plasmas*, vol. 30, p. 010501, 2023.

[3] C. Alappat, N. Meyer, J. Laukemann, T. Gruber, G. Hager, G. Wellein, and T. Wettig, "Execution-cache-memory modeling and performance tuning of sparse matrix-vector multiplication and lattice quantum chromodynamics on a64fx," *Concurrency and Computation: Practice and Experience*, vol. 34, p. e6512, 2022.

[4] Z. Dong, Y.-L. L. Fang, X. Huang, H. Yan, S. Ha, W. Xu, Y. S. Chu, S. I. Campbell, and M. Lin, "High-performance multi-mode ptychography reconstruction on distributed gpus," in *Proc. IEEE NYSDS*, 2018, pp. 1–5.

[5] V. T. Sabe, T. Ntombela, L. A. Jhamba, G. E. Maguire, T. Govender, T. Naicker, and H. G. Kruger, "Current trends in computer aided drug design and a highlight of drugs discovered via computational techniques: A review," *European Journal of Medicinal Chemistry*, vol. 224, p. 113705, 2021.

[6] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[7] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proc. NAACL-HLT*, 2019, pp. 4171–4186.

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Proc. NeurIPS*, vol. 32, 2019.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[10] "NVIDIA Data Loading Library," https://developer.nvidia.com/DALI.

[11] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and mitigating data stalls in dnn training," in *Proc. VLDB Endow.*, 2021, p. 771–784.

[12] J. Liu, B. Nicolae, D. Li, J. M. Wozniak, T. Bicer, Z. Liu, and I. Foster, "Large scale caching and streaming of training data for online deep learning," in *Proc. FlexScience*, 2022, pp. 19–26.

[13] J. Liu, B. Nicolae, and D. Li, "Lobster: Load Balance-Aware I/O for Distributed DNN Training," in *Proc. ICPP*, 2022, pp. 1–11.

[14] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning i/o," in *Proc. ACM/IEEE SC*, 2021, pp. 1–15.

[15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proc. IEEE CVPR*, 2009, pp. 248–255.

[16] M. Arif, K. Assogba, and M. M. Rafique, "Canary: fault-tolerant faas for stateful time-sensitive applications," in *Proc. ACM/IEEE SC*, 2022, pp. 568–583.

[17] A. Ali, H. Sharma, R. Kettimuthu, P. Kenesei, D. Trujillo, A. Miceli, I. Foster, R. Coffee, J. Thayer, and Z. Liu, "fairDMS: Rapid model training by data and model reuse," in *Proc. IEEE CLUSTER*, 2022, pp. 394–405.

[18] Y. Liu, Y. Sun, B. Xue, M. Zhang, G. G. Yen, and K. C. Tan, "A survey on evolutionary neural architecture search," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, pp. 550–570, 2023.

[19] H. Liu, B. Nicolae, S. Di, F. Cappello, and A. Jog, "Accelerating DNN Architecture Search at Scale Using Selective Weight Transfer," in *Proc. IEEE CLUSTER*, 2021, pp. 82–93.

[20] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proc. ISCA*, 2015, pp. 158–169.

[21] M. Kuchnik, A. Klimovic, J. Simsa, V. Smith, and G. Amvrosiadis, "Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines," in *Proc. MLSys*, vol. 4, 2022, pp. 33–51.

[22] D. Choi, A. Passos, C. J. Shallue, and G. E. Dahl, "Faster neural network training with data echoing," *arXiv preprint arXiv:1907.05550*, 2019.

[23] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu, "I/o characterization and performance evaluation of beegfs for deep learning," in *Proc. ICPP*, 2019, pp. 1–10.

[24] S. Pumma, M. Si, W. 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.

[25] Z. Zhang, L. Huang, J. Pauloski, and I. T. Foster, "Efficient I/O for neural network training with compressed data," in *Proc. IEEE IPDPS*, 2020, pp. 409–418.

[26] M. Arif, A. Maurya, and M. M. Rafique, "Accelerating performance of gpu-based workloads using CXL," in *Proc. FlexScience*, 2023, p. 27–31.

[27] M. Madhyastha, R. Underwood, R. Burns, and B. Nicolae, "Dstore: A lightweight scalable learning model repository with fine-grained tensor-level access," in *Proc. ICS*, 2023.

[28] E. Gianniti, L. Zhang, and D. Ardagna, "Performance prediction of gpu-based deep learning applications," in *Proc. IEEE SBAC-PAD*, 2018, pp. 167–170.

[29] S. Kaufman, P. Phothilimthana, Y. Zhou, C. Mendis, S. Roy, A. Sabne, and M. Burrows, "A learned performance model for tensor processing units," in *Proc. MLSys*, vol. 3, 2021, pp. 387–400.

[30] S. Fu, S. Gupta, R. Mittal, and S. Ratnasamy, "On the use of ML for blackbox system performance prediction," in *Proc. USENIX NSDI*, 2021, pp. 763–784.

[31] S.-M. Tseng, B. Nicolae, G. Bosilca, E. Jeannot, A. Chandramowlishwaran, and F. Cappello, "Towards portable online prediction of network utilization using mpi-level monitoring," in *Proc. Euro-Par*, 2019, pp. 47–60.

[32] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *Proc. ICLR*, 2017.

[33] S. Lym, D. Lee, M. O'Connor, N. Chatterjee, and M. Erez, "Delta: Gpu performance model for deep learning applications with in-depth memory system traffic analysis," in *Proc. IEEE ISPASS*, 2019, pp. 293–303.

[34] X. Y. Geoffrey, Y. Gao, P. Golikov, and G. Pekhimenko, "Habitat: A Runtime-Based computational performance predictor for deep neural network training," in *Proc. USENIX ATC*, 2021, pp. 503–521.

[35] A. Jajoo, Y. C. Hu, X. Lin, and N. Deng, "A case for task sampling based learning for cluster job scheduling," in *Proc. USENIX NSDI*, 2022, pp. 19–33.

[36] G. F. M. Yeung, *Proactive Interference-Aware Resource Management in Deep Learning Training Cluster*. Lancaster University (United Kingdom), 2022.

[37] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *Proc. IEEE/ACM MICRO*, 2015, pp. 62–75.

[38] D. Graur, D. Aymon, D. Kluser, T. Albrici, C. A. Thekkath, and A. Klimovic, "Cachew: Machine learning input data processing as a service," in *Proc. USENIX ATC*, 2022, pp. 689–706.

[39] W. Chen, S. He, Y. Xu, X. Zhang, S. Yang, S. Hu, X.-H. Sun, and G. Chen, "icache: An importance-sampling-informed cache for accelerating i/o-bound dnn model training," in *Proc. IEEE HPCA*, 2023, pp. 220–232.

[40] 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 *Proc. ISCA*, 2022, pp. 1042–1057.

[41] Y. Le and X. Yang, "Tiny imagenet visual recognition challenge," *CS 231N*, 2015.

[42] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proc. USENIX ATC*, 2020, pp. 219–233.