DynPipe: Toward Dynamic End-to-End Pipeline Parallelism for Interference-Aware DNN Training

Zhengyi Yuan, Xiong Wang, *Member, IEEE*, Yuntao Nie, Yufei Tao, Yuqing Li, *Member, IEEE*, Zhiyuan Shao, *Member, IEEE*, Xiaofei Liao, *Member, IEEE*, Bo Li, *Fellow, IEEE*, and Hai Jin, *Fellow, IEEE*

***Abstract*—Pipeline parallelism has emerged as an indispensable technique for training large deep neural networks. While existing asynchronous pipeline systems address the time bubbles inherent in synchronous architectures, they continue to suffer from *inefficiency* and *susceptibility* to *volatile* hardware environment due to their suboptimal and *static* configurations. In this article, we propose DynPipe, an *interference-aware* asynchronous pipeline framework to optimize the *end-to-end* training performance in highly *dynamic* computing environments. By characterizing the *non-overlapped* communication overheads and *convergence* rate conditioned on stage-wise staleness, DynPipe carefully crafts an optimized pipeline partition that harmonizes the hardware speed with statistical convergence. Moreover, DynPipe deploys a *non-intrusive* random forest model that utilizes runtime stage statistics to evaluate the impact of environmental changes, such as task interference and network jitter, on the training efficiency. Following the evaluation guidance, DynPipe adaptively *adjusts* partition plan to restore both intra and inter-stage load balancing, thereby facilitating seamless pipeline reconfiguration in dynamic environments. Extensive experiments show that DynPipe outperforms state-of-the-art systems, accelerating the time-to-accuracy by *1.5-3.4*×.**

***Index Terms*—Pipelined training, dynamic computing environment, end-to-end performance, model re-partition.**

I. Introduction

In recent years, large deep neural networks (DNNs), such as BERT [1], Transformer [2], GPT [3], and VGG [4], have achieved enormous success across various fields including computer vision, neural language processing, and recommendation systems. For example, Llama-3 [5] which redefines the boundaries of DNN models has over 405 billion parameters, a scale far beyond the capacity of commodity servers. Training these colossal DNNs necessitates a dedicated cluster of advanced accelerators (e.g., GPUs) over months of continuous computation. This ever-expanding model size underscores the need for more effective distributed training frameworks.

Distributed DNN training primarily falls into data parallel and model parallel frameworks [6]. Data parallelism distributes the training dataset across GPU workers with each maintaining a complete model replica, while model parallelism partitions model operators over workers [7]. However, as DNNs grow larger, data parallelism is hampered by the limited memory of GPUs, and model parallelism suffers from low resource utilization due to its sequential operation dependencies. Pipelined model parallelism (i.e., pipeline parallelism) has emerged as a viable solution to partition DNNs into stages and train them in a pipelined manner, allowing for an overlap of computation and waiting time of different batches [8]. In contrast to synchronous pipeline frameworks with intrinsic time bubbles [9], [10], asynchronous systems alleviate idle bubbles via one-forward-one-backward (1F1B) pipeline scheduling, thus facilitating stages to alternate between forward and backward passes without a strict synchronization barrier [11], [12], [13].

Though effective, current asynchronous pipeline systems orchestrate the training process typically following a *static partition plan* [9], [11]. However, computing environments, like clusters with GPU sharing [14] and cloud spot instances [15], are highly *dynamic*, subject to fluctuations from task preemption, external interruptions, network inconsistencies, internal GPU errors, etc. [16]. Even in the absence of explicit interferences, the hardware environment remains heterogeneous during the training of large-scale models, since involving more GPU workers in the training invariably increases the likelihood of encountering dynamic stragglers [17]. Such variability can drastically undermine the training efficiency of a preset model partition. Another pressing concern lies in the pipeline configuration that *myopically enhances* the hardware speed for reduced training iteration time. A finer-grained partition in asynchronous pipeline frameworks can indeed boost the training throughput, but it may also proportionally increase the stage staleness and memory consumption under 1F1B scheduling policy, potentially degrading the model's *statistical performance* and limiting the size of trainable models [18]. Given that the overall system performance is intricately linked to statistical efficiency, focusing solely on mitigating per-iteration time to expedite pipelined training may be misguided.

Received 24 February 2025; revised 15 July 2025; accepted 25 August 2025. Date of publication 3 September 2025; date of current version 26 September 2025. This work was supported by NSFC-RGC under Grant 62461160333. Recommended for acceptance by D.-N. Yang. *(Corresponding author: Xiong Wang.)*

Zhengyi Yuan, Xiong Wang, Yuntao Nie, Yufei Tao, Zhiyuan Shao, Xiaofei Liao, and Hai Jin are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab/Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: d202381426@hust.edu.cn; xiongwang@hust.edu.cn; nieyuntao@hust.edu.cn; taoyufei@hust.edu.cn; zyshao@hust.edu.cn; xfliao@hust.edu.cn; hjin@hust.edu.cn).

Yuqing Li is with the School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China (e-mail: li.yuqing@whu.edu.cn).

Bo Li is with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay Hong Kong (e-mail: bli@cse.ust.hk).

Digital Object Identifier 10.1109/TPDS.2025.3605491

This study aims to develop an optimal *end-to-end pipeline* framework for distributed DNN training, that can adapt to *changing computing environments*. Despite the progress in workload scheduling on GPU clusters, most works focus on course-grained allocations of compute resource for assorted machine learning jobs, with limited attention to the nuanced requirements of pipelined training DNN models [19], [20]. There have been attempts to provide dynamic model partition for pipeline parallelism, yet they often target for converging to a stable configuration, sidestepping the complexities of time-varying environments [12]. To optimize end-to-end training performance, it is imperative to strike a balance between the hardware speed and statistical efficiency, especially when dealing with environmental changes, e.g., task interruptions, GPU stragglers. This, however, is challenging for the following reasons.

First, an end-to-end pipeline formulation must accurately determine the wall-clock training time, which depends on both the iteration duration and total number of updates required for convergence. However, *convergence characterization* is intricately coupled with pipeline stage partition, as 1F1B scheduling induces a stage-wise model staleness, and together with dependencies between consecutive stages, significantly complicates the convergence analysis. Second, reducing per-iteration update time necessitates a precise evaluation of pipeline operations. Common pipeline frameworks often assume that *inter-stage communication* time is entirely *overlapped* by computation so as to simplify their pipeline model. Nonetheless, this assumption is usually at odds with real-world profiling measurements, where the communication overhead is comparable to computation time. Even worse, the way DNNs are partitioned affects both the training convergence and iteration time, suggesting that a holistic *system co-design* is essential for true end-to-end optimization. Third, fostering dynamic pipeline partition involves accommodating the interference from externally submitted tasks. With limited ability to intrusively access these tasks' user code, recalibrating the partition plan may suspend ongoing training sessions, adversely affecting training efficiency. Therefore, adapting model partitioning strategy to dynamic computing environments must operate in a *swift* and *non-intrusive* manner. Fourth, achieving elastic pipelined training typically requires saving and distributing checkpoints of the system state to facilitate layer migration and tensor re-partition. However, existing checkpointing schemes are primarily designed for data parallel systems [21] and do not support *reconstructing training pipeline without interruption* [22] effectively. Moreover, because asynchronous pipelines inherently contain fewer idle periods, checkpointing can significantly hinder the training progress.

To address these challenges, we propose DynPipe, an elastic pipeline framework, to enhance end-to-end training performance in dynamic computing environments. DynPipe excels by *subtly modeling* pipeline operations and multi-stage asynchronous executions, which enable us to accurately characterize the iteration time and comprehensively factor in the staleness for convergence estimation. Atop of these insights, DynPipe progressively forms partitioned stages and swaps stashed model parameters to CPU, from the input layer to the output to harmonize the hardware speed and statistical efficiency. As a result, we can remarkably promote end-to-end training efficiency. When hardware environment changes (e.g., external interruption, unexpected GPU stragglers) during training, DynPipe builds a light-weight random forest model to non-intrusively gauge the impact of task interference. Specifically, we only require the *runtime statistics* measured for each stage and worker (i.e., forward/backward pass time, GPU status, network bandwidth, DRAM Rx throughput) to evaluate the performance of any new model partition, which provides valuable guidance for subsequent partition plan adjustments. Therefore, DynPipe can *migrate* the workload from temporally straggling workers to expedite the training process. We have implemented DynPipe atop PyTorch with a *re-designed* communication interface to facilitate a *seamless* pipeline re-deployment. Our main contributions are summarized.

- We highlight the inefficiency and vulnerability to training environmental changes that plagues current pipeline systems, and hence propose DynPipe, a *dynamic* pipeline framework for training large-scale DNN models. To the best of our knowledge, this is the *first attempt* that enhances the end-to-end model performance in time-varying computing environments.
- DynPipe *quantitatively characterizes* the impact of stage-wise model staleness on training convergence, offering invaluable insights for devising an optimized end-to-end model partition that strikes a strategic balance between hardware speed and statistical efficiency. Also, DynPipe implements an *efficient* communication interference and model offloading, curtailing the non-overlapped communication overhead and GPU memory consumption.
- DynPipe builds a *non-intrusive* random forest that utilizes collected runtime statistics to assess the performance decline caused by exogenous task interference. On this basis, DynPipe develops an agile model re-partitioning strategy that encompasses *transparent* intra and inter-stage load balancing. By pipelining the migration operation and prioritizing training traffic, DynPipe achieves seamless model re-partition without disrupting the training workflow. Therefore, DynPipe bolsters a system resilience and efficiency in dynamic computing environments.
- Extensive experiments on real-world datasets show the superiority of DynPipe in model performance and interference alleviation. In particular, DynPipe accelerates the time-to-accuracy by *1.5-3.4*$\times$ than state-of-the-art systems.

II. Background and Motivation

We first introduce the preliminary of distributed DNN training and dynamic training scheduling, then present the motivation of our DynPipe.

A. Distributed DNN Training

As DNNs sacle in size, sheer volume of their model parameters balloons. Large-scale GPU clusters are essential for distributed DNN training [23], [24], which is typically divided into data parallel and model parallel systems [6].

Data parallelism is effective for small DNN models that can fit in GPU's physical memory (at tens of GB) [25]. As today's DNNs often require GPU memory in the hundreds of GBs or even TBs (e.g., Llama-2-70B [26], GPT-4 [27]), model parallelism becomes vital by partitioning the entire model into smaller stages, thus reducing the memory burden on any single worker [28]. Nonetheless, inherent sequential execution in vanilla model parallelism can lead to considerable idle time, up to 7% or more per stage [29], accumulating to *over 50%* if model split into eight stages, which in turn impairs training efficiency. Synchronous pipelined model parallelism is designed to process multiple minibatches concurrently to overlap computation and communication executions [9], [10]. Due to the synchronization barrier, it can incur notable peak memory consumption to store multiple copies of model parameters, and still introduces persistent time bubbles where GPUs lie idle. Asynchronous pipeline circumvents the synchronization delays by eschewing the barriers, allowing a minibatch of data to be processed via different versions of model parameters [11], [12]. Though effective, asynchronous systems bring in *stage-wise model staleness*, potentially slowing down the training convergence.

B. Dynamic Training Scheduling

More and more attentions have been drawn to accommodating dynamic computing environments, driven by involving DNN architectures [30], training strategies like adaptive layer freezing [31], GPU sharing [32], [33], and notably the impact of external task interference [34]. Recent studies reveal that GPU machines can experience preemption as frequently as every 15 minutes and once within a 24-hour period in cloud spot instances [21], exposing distributed DNN training to frequent interruptions from competing task submissions. Even in homogeneous hardware environments, some GPUs occasionally underperform due to factors like network jitter or GPU auto-throttling [35], underscoring the necessity of dynamic training management. A pipeline framework generally encompasses model profiling and training deployment. In the profiling phase, we run training iterations on a single GPU to collect critical data, such as the time of per-layer forward and backward passes, model size, gradients and activations, etc., all of which are instrumental for guiding pipeline deployment that follows. However, the volatile nature of computing environments can rapidly render initial profiling outdated, severely compromising the training efficiency. Hence, there is a pressing need to *re-balance workloads* across workers in line with changes in compute resources and task demands.

C. Motivation

1) Non-Negligible Inter-Stage Communication Overhead: The efficiency of pipelined training is tightly dependent on the accuracy of model profiling, based on which pipeline frameworks will optimize the model stage partition and assign each stage to designated GPU workers, as seen in Dapple [10], PipeDream [11], and Megatron [36]. During training, the transfer of essential tensors, including intermediate gradients and activations, is facilitated by inter-stage communication, which exhibits distinct characteristics across different DNN architectures. For instance, communication overhead is relatively uniform among consecutive layers in Transformer-based language models, whereas it is considerably skewed in large-scale CNN models due to the heterogeneity of model layers [37].

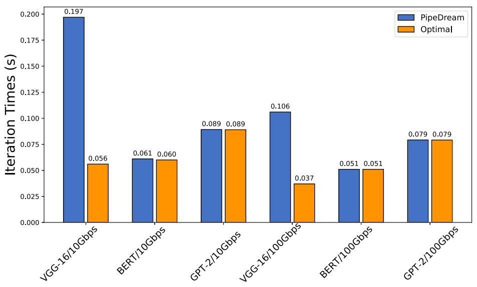


Fig. 1. Non-negligible communication: eight GPUs across two servers interconnected by 10/100 Gbps links.

A fundamental deficiency of existing pipeline frameworks is the *erroneous assumption* that the communication overhead can be completely overlapped by concurrent computations. Although this assumption simplifies pipeline formulation, it is inconsistent with real-world profiling data, leading to a suboptimal model partition, particularly for DNNs with heterogeneous architectures. To elaborate, when employing PipeDream to partition isomerous VGG-16 with 38 layers [4] and Transformer-based BERT into eight stages, Fig. 1 reveals a pronounced slowdown in what PipeDream predicted as the optimal partition for VGG-16 compared to the actual optimal plan. Conversely, partitions achieved by PipeDream for BERT and GPT-2 appear to maintain the highest efficiency. This discrepancy arises because the initial pipeline formulation fails to factor in inter-stage communication delays which are optimistically expected to overlap with computations, thus resulting in a model partition misaligned with actual operational context for heterogeneous DNN models. Our later analysis in Fig. 7 shows that the communication latency can only be completely offset if one stage's execution time notably exceeds other stage's combined execution and communication time, which however is uncommon as workloads across stages are often balanced to maximize the training throughput.

2) Suboptimal End-to-End Performance: Asynchronous pipeline parallelism can achieve higher training efficiency compared to the synchronous systems by utilizing 1F1B scheduling to allow each stage to alternate between forward and backward passes. While 1F1B effectively eradicates the idle time bubbles, it incurs a trade-off in stage-wise *model staleness* and additional *memory consumption*, since maintaining computational consistency requires storing multiple versions of model parameters to use the same model version for both forward and backward passes within each minibatch. These factors can adversely affect the statistical convergence rate and limit the size of trainable DNN models. Current pipeline frameworks mostly increase the granularity of stage partitions to optimize the time taken for per-iteration model update [11], [38], yet overlooking the impact of increased staleness and GPU footprint, which may negate the gains from expedited execution speed.

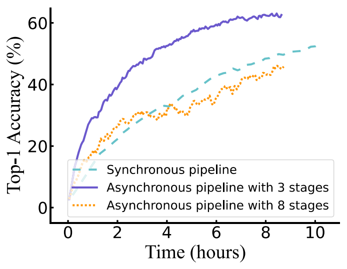


Fig. 2. Staleness impact on wall-clock time.

To show this, we train VGG-16 following both synchronous and asynchronous pipelines. Results in Fig. 2 reveal that although the asynchronous pipeline accelerates the initial training phase and exhibits faster per-iteration time compared to the synchronous method, it may require more time to reach the target accuracy (50%) as the number of pipeline stages increases. The root cause lies in the singular focus on optimizing hardware speed. Additionally, the average memory requirement also increases with more pipeline stages; for instance, an eight-stage setup requires 1.25 times more memory than a three-stage setup. For efficient end-to-end pipeline, it is imperative to balance the *statistical convergence and hardware speed*, while reducing the memory cost so as to enhance training performance.

3) Vulnerable to Dynamic Computing Environment: Pipeline parallelism often commits to a static model partition configuration. Given the fact that considerable time is spent on training large DNN models, which can span from tens of hours to several months [26], [27], the training process is *vulnerable to dynamic computing environments*, such as exogenous task interruptions during GPU sharing services, the use of spot instances, or network link jitter.

Existing pipeline frameworks lack the agility to adjust their model partition dynamically in response to such mid-course disturbances. Fig. 3(a) exhibits that dynamic preemption in spot instance services can severely slow down the training process. Concerning task interference, current strategies primarily focus on scheduling machine learning tasks within GPU clusters, where a global scheduler can access the user code of each task [39]. However, this assumption does not hold in training a specific DNN model, where incoming tasks that cause interruptions are opaque to the pipeline framework. Volatile environment caused by task interference may arise from various sources, such as contention over the GPU's memory bus that hampers memory access for the current task's streaming multiprocessors (SM), or over-allocation of SMs which leads to queuing and prolonged task latency. To understand the interference impact, we introduce exogenous task interference while training VGG-16 with PipeDream, recording PCIe write/read requests to BAR1, SM activity, and DRAM Rx throughput as shown in Fig. 3(b). Similar to spot instance, task interference delays training. However, our findings indicate that changes in GPU status correlate with the level of instance preemption or task interruption, which can provide valuable guidance for interference-aware pipeline design when intimate knowledge of external tasks is absent.

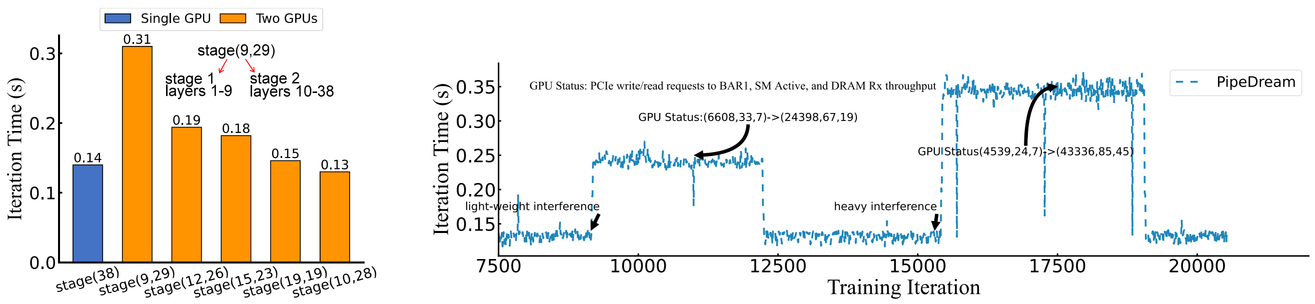
III. System Design

We propose DynPipe, an efficient pipeline framework, to enhance end-to-end DNN training performance in dynamic computing environments. DynPipe harnesses a precise *layer-wise profiling* to enable an optimal model partition and curtailing wall-clock training time. Fig. 4 shows the system architecture of DynPipe which is composed of the pipeline planner, global manager and execution engine.

Pipeline Planner: The pipeline planner embarks on a layer-wise model profiling, covering the time of layer's forward and backward passes, and size of gradients and activations. On this basis, it establishes an optimized model partition to harmonize the statistical training convergence with per-iteration execution time considering non-overlapped communication time. Once model stages are defined, they will be deployed to designated workers and start pipelined training under 1F1B policy.

Global Manager: The global manager's role lies in synthesizing information from all execution engine, and formulating a new partition plan through a lightweight re-partitioning algorithm. This re-calibrated plan is then communicated back to each scheduler, aimed at re-balancing the communication and computation workloads across GPU workers to adapt to dynamic environment caused by external task interference. For a precise evaluation of model performance during re-partition, we opt for Nsight Systems [40] rather than NVIDIA-smi command to ascertain GPU activity. *Co-located* with the final stage, global manager negates the need for creating extra communication process, as the collection of runtime statistics can be piggybacked by the transfer of forward activations.

Execution Engine: Each GPU worker features an adaptive execution engine that manages the entire training lifecycle, encompassing both forward and backward executions as well as on-the-fly model reconfiguration. During the forward pass, the engine receives activations from the upstream stage, executes local kernels, and forwards the results. The backward pass mirrors this process for intermediate gradients. To handle multiple minibatches in flight, the engine maintains up to S (the number of stages) weight versions in a GPU "weight cache", prefetching the latest versions from pinned host memory and evicting old ones to minimize device footprint. To enable model re-partition, the engine records per-layer forward/backward latency, SM utilization, DRAM throughput, and available memory every few hundred milliseconds. These statistics are sent to the global manager, which determines when the current partition is imbalanced. Upon receiving a new plan from the manager, the execution engine reconfigures the training online. It maintains an explicit graph representation of its model portion, rewrites this graph during migration, and packages the affected layers' weights, optimizer states, and random number generator seeds into checkpoints for transfer to their new hosts. Serialization and network transfer are pipelined by a priority scheduler that throttles checkpoint packets behind latency-sensitive activation and gradient traffic, thus preventing training slowdowns. By integrating scheduling, profiling, and migration into a single component, DynPipe achieves model re-partitioning with minimal communication and storage overhead, ensuring a bubble-free pipeline even in dynamic computing environments.



(a) Spot instance (b) Task interference

Fig. 3. Dynamic computing environment: spot instance and task interference.

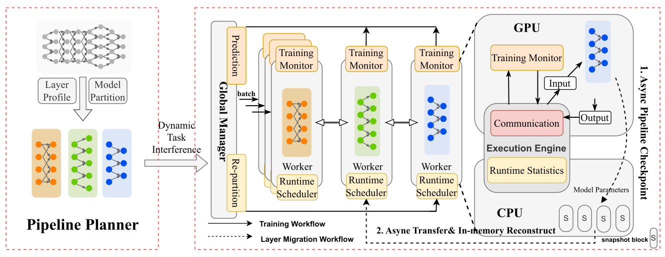


Fig. 4. System framework of DynPipe.

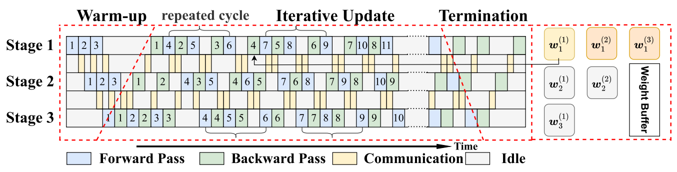


Fig. 5. Pipelined training decomposition.

IV. End-to-End Pipeline Planning

In this section, we elaborate the pipeline planning which includes an in-depth analysis of training convergence and a precise pipeline characterization. By doing so, DynPipe *proactively* enhances end-to-end model performance, ahead of environmental changes, and also *initializes* pipeline configuration for subsequent model re-partition.

A. Wall-Clock Time Characterization

Pipelined training generally unfolds across three distinct phases: warm-up, iterative update, and termination phases, as depicted in Fig. 5. The warm-up phase begins with a sequential insertion of stage-specific minibatches into the pipeline. Duration of this phase is negligible compared to overall training time, which encompasses hundreds or thousands of model updates per epoch. After the warming up, the training transitions into a continuous cycle of asynchronous model updates, termed iterative update phase. As an epoch draws to a close, training enters the termination phase, which enforces synchronized executions of forward and backward passes across all pipeline stages. Time of this closing phase is typically on par with that of the warm-up.

Therefore, strategic reduction of the training time hinges on curtailing the time expended during the *repeated cycles of model update*, which is determined by both the per-iteration duration and total number of updates. In particular, the time for a single model update, i.e., $\boldsymbol{w}^{(t)} \rightarrow \boldsymbol{w}^{(t+1)}$, consists of the computation (forward and backward passes) time for each stage and non-overlapping communication time. Regarding the number of model updates, it depends on the training convergence rate. By discerning these two critical factors, we can facilitate an end-to-end pipeline optimization.

B. Training Convergence

1) Stage-Wise Staleness: DynPipe employs 1F1B scheduling policy with weight stashing to preserve historical model weights during pipelined training [11], which ensures that each stage can compute activations and gradients using a consistent set of model parameters. For instance, when the entire model is partitioned into three stages with each designated to respective workers as in Fig. 5, stage 1, during the forward pass for minibatch 4, will utilize the model that were refreshed by minibatch 1 in its backward pass. Then, the stage retains the model in stasis until minibatch 4 stashes its backward pass. Only after the model parameters corresponding to minibatch 4 have been updated, the stashed stale model can be discarded and replaced with the refreshed one based on minibatch 4. Consequently, each pipeline stage is subject to a certain degree of staleness, with a delay that is proportional to the number of stages S, where $S = 3$ in this example. In broad strokes, the t-th model update can be described by:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \cdot \tilde{\nabla} f\left(w_1^{(t-S+1)}, w_2^{(t-S+2)}, \ldots, w_S^{(t)}\right), \tag{1}$$

where η is the learning rate [11]. Besides, $\boldsymbol{w}^{(t)} = \{w_1^{(t-S+1)}, w_2^{(t-S+2)}, \ldots, w_S^{(t)}\}$ refers to the entire model in which $w_i^{(t-S+i)}$ represents the stashed model weights for stage i with a stage-wise staleness $S - i$. Besides, $\tilde{\nabla} f$ denotes the stochastic gradient of the loss function $f(\cdot)$. Model staleness is a byproduct of 1F1B policy, which accelerates the hardware speed with mitigated per-iteration time, but can harm the statistical training convergence.

2) Convergence Result: Characterizing the convergence rate of DynPipe is challenging due to model partition and variable staleness across different stages. To facilitate convergence analysis, we make following widely-adopted assumptions [6].

Assumption 1 (Smoothness): Gradient of the loss function is L-smooth, i.e., $\|\nabla f(\boldsymbol{w}_1) - \nabla f(\boldsymbol{w}_2)\| \leq L\|\boldsymbol{w}_1 - \boldsymbol{w}_2\|$.

Assumption 2 (Bounded Moment): The stochastic gradient is bounded, i.e., $\mathbb{E}[\|\tilde{\nabla} f(\boldsymbol{w})\|^2] \leq \xi^2$.

Assumption 3 (Bounded Variance): The variance of stochastic gradient on a training sample x is bounded, i.e., $\mathbb{E}[\|\tilde{\nabla} f(\boldsymbol{w}, x) - \nabla f(\boldsymbol{w})\|^2] \leq \sigma^2$.

In essence, Assumption 1 quantifies the impact of model changes on the loss. Assumption 2 indicates that the second moment is upper capped. Meanwhile, Assumption 3 implies a finite bound for the stochastic gradient variance. Next, we present our main convergence result, while the proof is presented in Appendix A, available online.

Theorem 1: Under Assumptions 1–3 and $\eta L \leq 1$, it satisfies

$$\frac{1}{T}\sum_{t=0}^{T-1} \mathbb{E}[\|\nabla f(\boldsymbol{w}^{(t)})\|^2] \leq \frac{2(f(\boldsymbol{w}^{(0)}) - f(\boldsymbol{w}^{(T)}))}{T\eta} + 2\eta^2 S^3 L^2 \xi^2 + \frac{2\sigma^2}{B}. \quad (2)$$

Theorem 1 implies that despite the term $\frac{2\sigma^2}{B}$, convergence error is altered by the number of updates T and the number of stages S. Therefore, while model partition may promote the hardware speed with reduced per-iteration time, it often leads to a decline in the statistical performance.

Referring to the convergence error in (2), we postulate that the pipelined training achieves convergence when the error falls beneath a specified minimal threshold ϵ [41]. This condition is approximated as $\frac{2(f(\boldsymbol{w}^{(0)}) - f(\boldsymbol{w}^{(T)}))}{T\eta} + 2\eta^2 S^3 L^2 \xi^2 \leq \epsilon$ by regarding $f(\boldsymbol{w}^{(T)}) = 0$. Then

$$T \geq \frac{2f(\boldsymbol{w}^{(0)}))}{\eta(\epsilon - 2\eta^2 S^3 L^2 \xi^2)}. \quad (3)$$

Determination for the required model updates is hindered by the presence of possibly unknown variables, like the smoothness constant L and second moment ξ^2. However, these variables can be estimated based on their definitions. For example, one can run the pipelined training for a few iterations, and measure the norm of gradient difference $\frac{\|\nabla f(\boldsymbol{w}^{(t)}) - \nabla f(\boldsymbol{w}^{(t-1)})\|}{\|\boldsymbol{w}^{(t)} - \boldsymbol{w}^{(t-1)}\|}$ in each iteration, which serves as the empirical samples for calculating the smoothness L. A similar method can be applied to estimate the second moment ξ^2 as well. In summary, an efficient model partition plan should balance the convergence error and hardware speed to optimize the end-to-end training performance.

C. Efficient Pipeline Plan

So far, we have analyzed the convergence error to obtain the required update iterations. The remaining issue is to characterize the time for a single update.

1) Accurate Pipeline Profile: Existing pipeline frameworks often operate under unrealistic assumption that inter-stage communication time is entirely overlapped, and use a recursive timeline to calculate per-iteration time. However, Fig. 1 evidences that communication overhead for many large models is generally

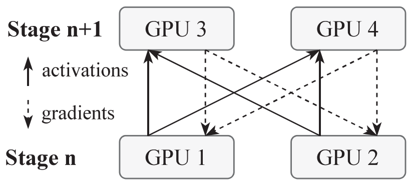


Fig. 6. An example of inter-stage communication.

substantial. Hence, we need to precisely formulate the pipeline operation ahead of devising model partition.

Consider that partitioning a DNN model with M layers into S stages, which will be deployed to N identical GPU workers. During training, each input minibatch progresses sequentially through all M layers in the forward pass and the procedure is reversed for the backward pass, enforcing *twice inter-stage communications* within one model update. As illustrated in Fig. 6, we consider the case where stages n and $n+1$ are assigned to two GPU workers for their intra-stage parallelization, which could involve data, tensor parallelization, or both. Then, activations from GPU 1 are split into two equal shares along the batch size axis, or are sharded, and then forwarded to stage $n+1$ through all-to-all communication, say GPU 3 might receive two shares of activations and merges them to finalize its forward pass. While GPUs assigned to stage $n+1$ will split the gradients accordingly, dispatching them to CPUs 1 and 2 also via all-to-all communication upon the completion of their backward pass. A significant advantage of intra-stage parallelization is the flexibility it affords in dynamically adjusting *data and tensor distribution* within stage, like more activations can be assigned to GPU 3 when GPU 4 is overloaded, to be elaborated in Section V-B.

2) Duration of Update Iteration: We now characterize the per-iteration time of a repeated cycle in Fig. 5, given that the model has been partitioned into S stages. Suppose we have identified the partition for initial l layers, which are split into s stages and assigned to the first i GPU workers with the last stage allocated r GPUs for intra-stage parallelization. The per-iteration time induced by such plan is denoted by $W(l, s, r, i)$. Our goal is to determine $W(S) \triangleq \min_{1 \leqslant r < N} W(M, S, r, N)$, which corresponds to the iteration time when entire M layers are partitioned into S stages and deployed to N workers. DynPipe devises dynamic programming to resolve the optimal pipeline partition, while before that the per-iteration time should be provided. With 1F1B scheduling, each stage alternates between forward and backward passes once receiving essential activations and gradients from preceding and subsequent stages.

We first specify the per-iteration time $W(2)$ for a simple two-stage scenario depicted in Fig. 7, and then extend the result to general model partition. Under 1F1B scheduling, if the computation time of Stage 2 is less than the combined computation and communication time of Stage 1, denoted as case 1, duration of a repeated cycle is determined by the inter-stage communication time $c_{1,2}$ (includes both the forward activation transfer from Stage 1 to 2 and the backward gradient transfer from Stage 2 to 1) and their average computation time, i.e.,

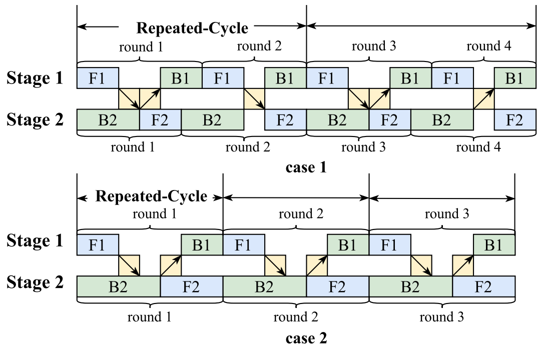


Fig. 7. Per-iteration time of two-stage partition.

$W(2) = \frac{c_{1,2}+p_{1,f}+p_{1,b}+p_{2,f}+p_{2,b}}{2}$ where $p_{1,f}, p_{2,f}$ and $p_{1,b}, p_{2,b}$ are the forward and backward pass time, respectively. If the situation is reversed, as in case 2, per-iteration time becomes $W(2) = p_{2,f} + p_{2,b}$. That is,

$$W(2) = \begin{cases} \frac{c_{1,2}+p_{1,f}+p_{1,b}+p_{2,f}+p_{2,b}}{2} & \text{case 1,} \\ p_{2,f} + p_{2,b} & \text{case 2.} \end{cases} \tag{4}$$

For general pipeline partition with $S > 2$ stages, we *iteratively split* the model by treating the first $S-1$ stages a unified entity (merged stage), in tandem with the last stage. As a result, we can apply the principle of (4) to derive transition equations as we progressively move the partition boundaries from the input toward the output layer:

$$W(M,S,r,N) = \min_{l',r'} \begin{cases} \frac{c_{l',l'+1}+W(l',S-1,r',N-r)+W_{l'+1\to M}}{2} & \text{case 1,} \\ W_{l'+1\to M} & \text{case 2.} \end{cases} \tag{5}$$

Here, $W(l', S-1, r', N-r)$ represents the computation time of the first (merged) stage where stage $S-1$ is assigned to r' workers, akin to $p_{1,f} + p_{1,b}$ in (4). Note that $W(l', S-1, r', N-r)$ also depends on intra-stage parallelization, including mesh connectivity and data/tensor distribution among the r' workers. Besides, $W_{l'+1\to M}$ concerns the assembly of the last stage spanning from layer $l'+1$ to output layer M, and this stage is assigned to r workers. Specifically, $W_{l'+1\to M}$ encompasses the time required for forward and backward passes at each layer. In other words,

$$W_{l'+1\to M} = \frac{\sum_{o=l'+1}^{l}(p_o^f + p_o^b) + p_{All-Gather}}{r_{data}\cdot r_{tensor}} + p_{All-Reduce}, \tag{6}$$

where p_o^f, p_o^b specify the per-layer forward and backward pass time. Terms r_{data} and r_{tensor} represent the degrees of intra-stage data and tensor parallelism, respectively, whose product satisfies $r_{data}\cdot r_{tensor} = r$, while $p_{All-Gather}$ and $p_{All-Reduce}$ denote the activation all-gather time and gradient all-reduce time, respectively. Inter-stage communication overhead is captured by:

$$c_{l',l'+1} = \frac{A_{l',l'+1} + G_{l'+1,l'}}{r' r b_{S-1,S}}, \tag{7}$$

Algorithm 1: Pipeline Planning.

```
Require: GPU number $N$, layer number $M$, model profile
  $\boldsymbol{G}_M$, network bandwidth $\boldsymbol{b} = \{b_{i,j}\}_{i,j=1}^N$
1:  $Rec \leftarrow \emptyset$
2:  for $S = 1, \ldots, N$ do
3:    $IterNumber \leftarrow$ CalcIteration($S$)    ▷Call for (3)
4:    $Rec' \leftarrow \emptyset$
5:    for $r = 1, \ldots, N$ do
6:      for $submesh \in$Mesh(r) do
7:        $W(M,S,r,N)$, $Partition'$, $submesh \leftarrow$
          Partition($\boldsymbol{G}_M, M, S, r, N, \boldsymbol{b}$)    ▷Dynamic
          programming
8:        Add $\{W(M,S,r,N), Partition', submesh\}$
          to $Rec'$
9:    $W(S) \leftarrow \min(Rec')$    ▷Iteration time for $S$
      stages
10:   $WallClockTime(S) \leftarrow IterNumber \times W(S)$
11:   Set $Partition(S)$ with $WallClockTime(S)$
12:   Add $\{WallClockTime(S), Partition(S)\}$ to $Rec$
13: $WallClockTime \leftarrow \min(Rec)$
14: Set $PartitionPlan$ with $WallClockTime$
15: return$PartitionPlan$
```

in which $A_{l',l'+1}$ is the activation size of layer l' and $G_{l'+1,l'}$ denotes the gradient size of layer $l'+1$. Term $b_{S-1,S}$ reflects the minimum network bandwidth between workers designated to the last two stages, which are divided across r' and r workers, respectively. Via an iterative adjustment for partition boundaries, we can minimize the per-iteration time $W(S)$ when the number of stages S is given.

3) Model Partition for End-to-End Pipeline: End-to-end optimal planning encompasses both the model partitioning strategy and the mapping of partitioned stages to GPU workers (i.e., intra-stage parallelization), which are inherently interdependent. To address this, we conceptualize the problem as a hierarchical optimization task. In the upper layer, DynPipe aims to optimize the pipeline parallelization latency by determining the optimal way to partition the model and GPU cluster into stages and worker meshes, and then pairing them as stage-mesh combinations. In the lower layer, DynPipe focuses on reducing the cost of executing a model stage on a specific worker mesh by optimizing the data and tensor parallelism plan. Hence, upper layer optimization relies on the stage-mesh costs provided by the lower layer. Given identical GPU workers, evenly distributing data or stage sharding proves to be the most effective strategy. To streamline the optimization space, we consider the communication demands of data and tensor parallelism, which generally require significant network bandwidth, thus confining these worker groups in the lower layer to a single physical machine. Take a machine containing eight GPUs as an example, the possible allocations for (data parallelism, tensor parallelism) can be {(1,8),(2,4),(4,2),(8,1)}.

In addressing the upper-layer problem, while we have characterized the convergence error and iteration duration, a closed-form wall-clock time still remains elusive. Combining with

(3), we recognize that both the total number of iterations and per-iteration time are influenced by the number of pipeline stages S. To determine the most effective model partition, DynPipe traverses all feasible stage numbers $S \in [1, N]$ where N is the number of GPU workers. For each S, DynPipe computes the required iterations $T(S)$ based on (3) and per-iteration time $W(S)$ based on (5). Consequently, the wall-clock training time is given by $T(S)W(S)$. A thumb of rule to *narrow* the searching space is to gradually increase S and then halt upon identifying a turning point. For the lower-layer problem, DynPipe leverages Alpa's operator optimizer [37] to resolve it.

The pipeline partition roadmap is formally encapsulated as Algorithm 1. At a high level, we will explore potential partition plans through an iterative adjustment of the partition boundaries (Lines 2-11). Simultaneously, we conduct a detailed exploration to identify the optimal intra-stage parallelism strategy in the lower layer (Lines 7-8). Finally, we choose the plan that results in the highest end-to-end training efficiency (Lines 13-14).

4) Weight Cache and Model Stage Swap: To ensure the correctness of asynchronous training, any pipeline system needs to use the same model parameters for both the forward and backward passes of a given minibatch. As illustrated in Fig. 5, when minibatch 4 initiates its backward pass, the model must revert to the parameter version that was updated following the completion of minibatch 1. Consequently, as the number of pipeline stages increases, devices responsible handling earlier stages must maintain multiple copies of the stage parameters, which significantly constrains the capability to train large-scale DNN models.

We observed that the model parameters for minibatch 4 were not required until its backward pass. This observation allows us to temporarily transfer these parameters to the more capacious CPU memory, swapping them back to the GPU only when necessary. A similar approach can be applied to other model versions. In this approach, DynPipe initially stores all model parameters in CPU memory. During training, based on the current GPU footprint, execution engine proactively prefetches the required model parameters from CPU memory into the GPU's weight cache, which can be synchronized with the execution of previous model states to minimize I/O delays. Parameters that are no longer needed are replaced as necessary. As a result, we can remarkably alleviate GPU memory pressure, ensuring more balanced memory usage across GPU workers and facilitating the training of larger DNN models.

V. Runtime Pipeline Adjustment

Based on the optimal static partition in Section IV, DynPipe dynamically re-balances communication and computation workloads across workers when encountering changes in computing environments. This is achieved by adjusting the distribution of training data and activations within a stage or by performing layer migrations between adjacent stages.

A. Non-Intrusive Interference Evaluation

Global manager of DynPipe is tasked with swiftly adjusting partition plan to accommodate dynamic computing environments. A primary challenge is the evaluation of re-partition performance, since pipeline stages may be co-located with externally submitted tasks, whose characteristics [43] resemble a closed box to DynPipe.

TABLE I
Task Interference Impact on Training Performance

Interference	Batch Size	Time Ratio	GPU Status Change
ResNet-50 [41]	32	1.1	(620,33,7) $\rightarrow$ (810,67,8)
2*ResNet-50	64	1.8	(620,33,7) $\rightarrow$ (954,72,10)
3*VGG-19 [4]	64	3.2	(620,33,7) $\rightarrow$ (973,73,31)

Evaluating the execution of a GPU task is non-trivial. Rapid, non-intrusive repartition remains difficult. Current systems like Middleware [44] and Horus [45] still require deep code inspection, collecting per-task FLOPs, operator counts, batch sizes, and other profiled details, so they cannot adapt dynamically without accessing the user's code. A common metric for assessing GPU status is SM utilization, which measures the percentage of SMs that are actively engaged, i.e., executing at least one warp. However, this metric alone is unable to provide a comprehensive view of GPU utilization, as an SM may be considered active even when only a minimal fraction of its resources are in use. Additionally, the data transfer workload between the GPU and CPU can affect overall GPU utilization. Bottlenecks in preprocessing input data on host CPUs can also cause GPUs to remain idle as they wait for data. Likewise, overhead from inter-machine communication also limits the training throughput, resulting in underutilized GPUs. To understand the interference impact, we co-locate a VGG-16 training task with assorted combinations of DNN workloads, differing in type and quantity, on a single GPU. Table I documents the time these runs take relative to isolated VGG-16 training, along with GPU status changes, including PCIe write/read requests to BAR1, SM Active, and DRAM Rx throughput. Our findings indicate that GPU status varies with the level of co-located workload, which can provide valuable guidance for interference-aware pipeline design when intimate knowledge of external tasks is absent.

Building on this foundation, we develop a lightweight and non-intrusive random forest model [46] that leverages *runtime statistics* from each worker, like GPU status, network bandwidth and data bus conditions, to predict the interference impact caused by co-located tasks. The model, constructed on a custom dataset offline, first captures the solo runtime of a typical GPU training task. We then expose the task to various levels of external interference, using the per-iteration time ratio to solo runtime as the ground truth, GPU status and task characteristics as input features to refine the random forest. During pipelined training, we assess new partition plans using random forest and collected statistics to optimize model partitions in response to interference. Fig. 8 shows an elementary overview of the interference prediction with a total of 32 features. Specifically, the diagram shows one representative decision tree (the forest contains 128 such trees) and traces the path that a particular runtime sample would follow from the root to a leaf. Each internal node lists the feature being tested, e.g., "GR Active", "SYS Clock Freq", or "Async Compute throughput", along with the corresponding split threshold. The two branches represent the standard decision-making process in random forests: "feature $\leq$ threshold" (left) and "feature $>$ threshold" (right). The

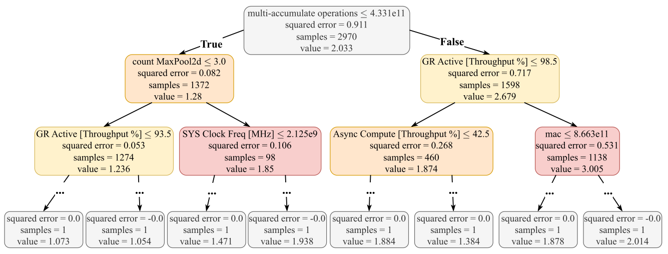

multi-accumulate operations ≤ 4.331e11
squared error = 0.911
samples = 2970
value = 2.033
True
False
count MaxPool2d ≤ 3.0
squared error = 0.082
samples = 1372
value = 1.28
GR Active [Throughput %] ≤ 98.5
squared error = 0.717
samples = 1598
value = 2.679
GR Active [Throughput %] ≤ 93.5
squared error = 0.053
samples = 1274
value = 1.236
SYS Clock Freq [MHz] ≤ 2.125e9
squared error = 0.106
samples = 98
value = 1.85
Async Compute [Throughput %] ≤ 42.5
squared error = 0.268
samples = 460
value = 1.874
mac ≤ 8.663e11
squared error = 0.531
samples = 1138
value = 3.005
squared error = 0.0, samples = 1, value = 1.073
squared error = -0.0, samples = 1, value = 1.054
squared error = 0.0, samples = 1, value = 1.471
squared error = -0.0, samples = 1, value = 1.938
squared error = 0.0, samples = 1, value = 1.884
squared error = 0.0, samples = 1, value = 1.384
squared error = 0.0, samples = 1, value = 1.878
squared error = -0.0, samples = 1, value = 2.014

Fig. 8. Random forest for interference level prediction.

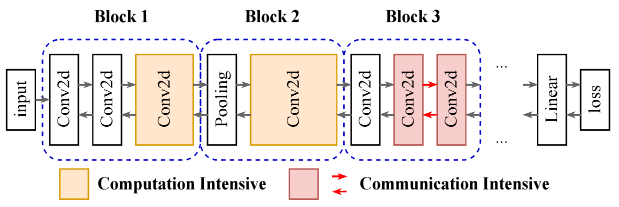


Fig. 9. Model condensation into blocks.

small grey box under every node reports the squared-error that would remain if the tree stopped splitting there, the number of training samples that reached the node, and the current mean value of the target variable (the iteration-time slow-down ratio). Additionally, we explore alternative models, including linear regression and neural networks. Our results indicate that the random forest model, when executed on CPUs, surpasses other models in terms of prediction error and computation efficiency.

B. Adaptive Model Re-Partition

We next elaborate the model re-partition when external tasks or GPU stragglers interrupt current training session, which involves the interplay between the global manager and execution engine, as illustrated in Fig. 4.

1) Model Condensation: Halting and restarting pipelined training in reaction to dynamic environment present considerable obstacles due to the complexity of modern DNN models. For instance, AmoebaNet [47], with its astounding 4280 layers, makes model re-partition and pipeline re-deployment from scratch a daunting task that requires a rapid solution, say within seconds. DNNs typically consist of a variety of layers, such as Relu, BatchNorm, Conv2d, FC layers and AveragePool, among others. Notably, Conv2d layers have disproportionately longer computation time than other types, resulting in a skewed distribution of layer execution time across the model. By cataloging layers according to their execution time and pinpointing the most time-intensive ones, the model can be segmented into *separate blocks*, each functioning as a standalone unit, as depicted in Fig. 9. This condensation, premised on the block of collective layers, aids in devising lightweight model re-partition strategies by considerably narrowing the solution space. Given structural nuances of DNN models, the loss incurred by model condensation is negligible when compared to the brute-force searching method, yet it is substantially more time-efficient.

2) Re-Partition Plan: Unlike static planning, presence of stragglers introduces heterogeneity into the training environment, making uniform data and tensor distribution within a model stage impractical. This heterogeneity expands the search space for mapping model stages to worker meshes, complicating direct application of Algorithm 1 to generate an effective pipeline plan. However, since the number of GPU stragglers typically represents a small fraction of the total workforce, we can begin with an existing optimal solution and incrementally refine it, which involves evaluating the improvements by swapping adjacent layers or adjusting intra-stage parallelization to approximate a near-optimal solution.

Under 1F1B policy, the training time is essentially reduced to the duration of iterative update phase, i.e., repeated cycles in Fig. 5. By harnessing the computation and communication time of each stage, we can precisely determine the per-iteration time by pinpointing the core cycle structure, which enables us to *simulate the performance* of a partition plan without the need for actual deployment. Building on this principle, DynPipe implements a three-step model re-partition strategy, described in Algorithm 2, to achieve a balanced workload across pipeline stages. Initially, DynPipe identifies pivotal top_k layers in the DNN model according to their present communication and computation time. Following the identification, DynPipe sequentially condenses the model, grouping consecutive layers into blocks that align with identified critical layers (Lines 3-6). The final step entails adjusting pipeline partition based on the condensed model. Specifically, DynPipe exhaustively explores the simplified model structure to locate the most effective partition boundaries and the load-balanced batch size or tensor sharding for intra-stage parallelization, where the evaluation of potential re-partition plans utilizes a tailored random forest to forecast the stage performance when they are co-located with interrupting tasks on designated GPUs. Overall, for each partitioning plan of the condensed model, Algorithm 1 iterates through all data parallel and tensor parallel worker groups, simulates the actual execution using a random forest for each combination, and evaluates the results. Performance metrics are then synthesized via an evaluation simulator to access the overall training efficiency and aids in selecting the optimal model re-partition plan (Lines 7–15).

Out-of-memory (OOM) issues can arise from task interference. To mitigate this, DynPipe continuously monitors each worker's real-time memory capacity and integrates this data into its migration logic. If a GPU worker reports a sudden decrease in free memory, an early warning of an impending OOM, the execution engine promptly throttles minibatch ingestion, evicts the least-recently-used weight snapshots from the GPU cache to host memory, and may trigger a rapid re-partition to migrate layers from the affected GPU before an allocation failure occurs. With checkpoint serialization pipelined and transmission prioritized below training traffic, these measures is executed within a few hundred milliseconds, effectively preventing CUDA OOM kills while sustaining overall training performance.

C. Transparent Layer Migration

Upon devising an enhanced re-partition plan, the global manager issues corresponding adjustment directives to each affected stage, coordinating the execution engine to transfer layer states among designated GPUs. The execution engine them will serialize layer states (e.g., model parameters, random

Algorithm 2: Model Re-Partition.

```
Require: Per-layer forward and backward pass time $\boldsymbol{p}^f = \{p_l^f\}_{l=1}^M, \boldsymbol{p}^b = \{p_l^b\}_{l=1}^M$, activation size $\boldsymbol{a} = \{a_l\}_{l=1}^M$, layer features $\boldsymbol{lf} = \{lf_l\}_{l=1}^M$, GPU status $\boldsymbol{s} = \{s_i\}_{i=1}^N$, network bandwidth $\boldsymbol{b} = \{b_{i,j}\}_{i,j=1}^N$
1:  $\boldsymbol{p}^{f'}, \boldsymbol{p}^{b'}, \boldsymbol{a}' \leftarrow$ ModelCondense($top\_k$)
2:  $RepartitionPlan \leftarrow$ Repartition($\boldsymbol{s}, \boldsymbol{p}^{f'}, \boldsymbol{p}^{b'}, \boldsymbol{a}', \boldsymbol{lf}$)
3:  function ModelCondense($top\_k$)
4:    $TopkLayer \leftarrow sorted(top\_k)$
5:    for $l \in TopkLayer$ do
6:      Add sum($\boldsymbol{p}^f[l-1:l]$), sum($\boldsymbol{p}^b[l-1:l]$) and sum($\boldsymbol{a}[l-1:l]$) to $\boldsymbol{p}^{f'}$, $\boldsymbol{p}^{b'}$ and $\boldsymbol{a}'$, respectively
7:  function Repartition($\boldsymbol{s}, \boldsymbol{p}^{f'}, \boldsymbol{p}^{b'}, \boldsymbol{a}', \boldsymbol{lf}$)
8:    if no repartition then
9:      return
10:   $Rec \leftarrow \emptyset$
11:   for $cm \in$ Combine($PartitionBoundary$) do
12:     $StageInfo \leftarrow$ GenStageInfo($cm, \boldsymbol{lf}$)
13:     $Straggle \leftarrow$ RandomForest($[\boldsymbol{s}, StageInfo]$)
14:     Add Simulator($Straggle, \boldsymbol{p}^{f'}, \boldsymbol{p}^{b'}, \boldsymbol{a}'$) to $Rec$
15:   Set $NewPartitionPlan$ with min($Rec$)
16:   return $NewPartitionPlan$
```

seed, optimizer states) into checkpoint files for their distribution to the specified stage.

1) Model Consistency: To maintain a consistent model, DynPipe pauses all parameter updates during a layer migration and buffers incoming gradients. Once the latest parameters are transferred, the execution engine swaps them in and gradually integrates the cached gradients into the subsequent optimizer steps. Take Fig. 5 as an example. When Stage 1 handles the backward for minibatch 4 and is instructed to transfer Layer 4 to the GPUs hosting Stage 2, it first halts local parameter updates while continuing to handle the forward and backward passes for minibatches already in the pipeline. Instead of applying their gradients, Stage 1 accumulates them locally. The version of Layer 4's weights currently being used by minibatch 7 is then serialized and sent to the destination worker. After the transfer is complete, Stage 2 instantiates Layer 4. Stage 1 then flushes the buffered gradients for that layer (e.g., gradients from $\boldsymbol{w}_1^{(1)}$, $\boldsymbol{w}_1^{(2)}$, $\boldsymbol{w}_1^{(3)}$) to the same destination worker. Only then are all stages permitted to proceed with the next optimizer step. This ensures every replica of Layer 4 is updated from the same set of gradients. In other words, the three transferred gradients are successively accumulated during the next three backward pass updates. The migration process concludes once Stage 2 successfully loads the layer's parameters and receives the associated gradients. Synchronization is ultimately achieved when each stage applies its accumulated gradients to update its respective parameters.

2) Asynchronous Checkpointing Execution: DynPipe breaks down the checkpoint snapshot and transfer into multiple steps, which execute in an asynchronous manner as shown in Fig. 10. They are (1) Serialization: checkpoint is serialized into byte streams and saved in CPU memory; (2) Distribution: DynPipe distributes checkpoints among workers several iterations behind ongoing training, allowing independent, asynchronous checkpoint transfer without data dependencies. During serialization, DynPipe segments the checkpoint stream into fixed-size chunks, which are then transferred *in parallel* during distribution. That is, they operate in a full parallelism.

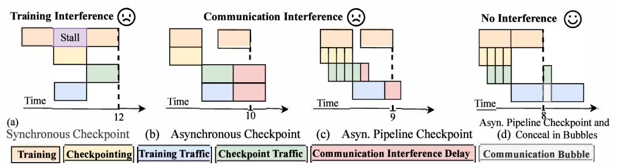

Training Interference ☹
Stall
Time
12
(a) Synchronous Checkpoint
Communication Interference ☹
Time
10
(b) Asynchronous Checkpoint
Time
9
(c) Asyn. Pipeline Checkpoint
No Interference ☺
Time
8
(d) Asyn. Pipeline Checkpoint and Conceal in Bubbles
Training
Checkpointing
Training Traffic
Checkpoint Traffic
Communication Interference Delay
Communication Bubble

Fig. 10. Pipelining checkpointing with computation.

3) Conceal Migration Within Bubbles: Checkpoint transfer is communication-intensive and will compete with inter-stage training traffic (activations/gradients) for limited network bandwidth. To minimize training disruptions, we design a flow control protocol to *prioritize training traffic* over checkpoints by using a priority queue in the OS network stack. Specifically, DynPipe employs `SO_PRIORITY` socket option to tag checkpoint packets for low-priority routing, while untagged training traffic enters the higher-priority queue which can preempt checkpoint to be dispatched to network interface card (NIC) buffer first. We find that flow control efficiency is sensitive to the checkpoint queue size $s_q^{(2)}$. Suppose that the arrival of both training and checkpoint packets follows a Poisson distribution with rates λ_1 and λ_2, respectively. Also, NIC processes packets at an exponential rate μ, functioning as a singular service channel. Based on queueing theory [48], when we reach steady state, then:

$$L_q^{(1)} = \frac{\lambda_1 \rho}{\mu - \lambda_1}, \; L_q^{(2)} = \frac{\lambda_2 \rho}{(\mu - \lambda_1)(1-\rho)}, \tag{8}$$

where $\rho = \frac{\lambda_1}{\mu} + \frac{\lambda_2}{\mu}$, and $L_q^{(1)}, L_q^{(2)}$ are the queue lengths of training and checkpoint packets, respectively. If queue size $s_q^{(2)}$ is too small, excess checkpoint packets will be either dropped or pushed back to the upper transport layer, potentially misleading transport layer's congestion and traffic control, while a too large $s_q^{(2)}$ can impair the preemptive efficiency, leading to training packets blocked by checkpoint data. Ideally, we can configure $s_q^{(2)} = L_q^{(2)}$, which is approximately 16 under 10Gbps network bandwidth. As a result, DynPipe achieves to conceal checkpoint distribution within idle bubbles without interfering with training communications.

Remark: We emphasize that our DynPipe is equally applicable to synchronous pipelined training systems such as PipeDream-Flush [18], Deepspeed [49] and Megatron [36]. This is because the communication overhead analysis and dynamic model re-partition are relevant across both synchronous and asynchronous setups. However, our primary focus on asynchronous pipelines stems from their inherent efficiency, since they eliminate the periodic idle bubbles that are typically observed in synchronous training schemes.

VI. System Implementation

We develop DynPipe atop PyTorch [50] with over 8000 lines of code, where the communication module and framework plugins are implemented using Python and GoLang.

Pipelined Training: To accommodate seamless model repartition amid unexpected interruptions, each scheduler maintains an exhaustive graph structure of the DNN model, including network structure declarations and tensor transfer procedures. Moreover, the scheduler is equipped with a streamlined parser that translates declarations into PyTorch Module commands. When model re-partition is necessary, the corresponding stage simply updates its network structure declaration to seamlessly spawn a fresh stage.

Communication Module: Current pipeline frameworks, such as PipeDream [11] and vPipe [12], employ a fixed round-robin method to distribute intra-stage parallelization workloads, sequentially routing minibatches in a predetermined order set before training starts. DynPipe enhances intra-stage parallelization by slicing tensors along different dimensions, offering more flexibility and adaptive load balancing among GPU workers compared to the rigid round-robin method. Additionally, these frameworks often struggle with workload variations during training.

To address this limitation, we have enhanced the communication module to enable user-controlled dispatching of minibatches among intra-stages. Specifically, tensors are sliced along batch or tensor dimensions, and these slices are then dispatched in parallel through a single asynchronous event loop within a custom ProcessGroupDyn. This approach allows multiple send/receive operations to progress concurrently, reducing per-stage latency and decreasing the number of communication threads from $O(G)$ to $O(1)$ where G is the number of transferred tensors. Our revised communication interface is fully compatible with PyTorch and existing communication backends, such as Gloo.

Model Re-partition: To accurately evaluate model performance after re-partitioning, we utilize Nsight Systems [40] rather than the NVIDIA-smi command for monitoring GPU activity. Also, the checkpointing module employs a priority-based queue within the kernel network stack to manage outgoing traffic from both training and checkpointing processes.

VII. Evaluation

In this section, we evaluate the performance of DynPipe in terms of the end-to-end performance and pipeline re-partition for different DNN models on real-world datasets.

A. Evaluation Setup

1) Hardware Setup: We conduct experiments to evaluate DynPipe using two representative GPU clusters.

- *Cluster A:* This cluster is equipped with two servers, each with six NVIDIA RTX 4090 GPUs (24 GB memory). Servers are interconnected via a 100 Gbps Mellanox ConnectX-5 InfiniBand adapter to facilitate communication.
- *Cluster B:* Comprising eight servers, each server in this cluster hosts four NVIDIA A10 GPUs. This configuration totals 32 GPUs across the cluster, with each GPU boasting 24 GB of memory, interconnected via NVLink. Communication between the servers is facilitated by a network link that provides a bandwidth capacity of 32 Gbps.

We employ PyTorch [50] version 1.7.1 and CUDA version 11.3, with Gloo as the communication backend. Besides, we train each DNN model using the SGD optimizer.

2) Models and Datasets: We identify three exemplary DNN models spanning various domains to evaluate DynPipe: 1) BERT-48 (680 M parameters) [1], 2) VGG-16 (110 M parameters) [4], and 3) GPT-2 1.7B/12B (1.7B/12B parameters) [51]. BERT-48 and GPT-2 1.7B/12B are trained on BookCorpus text dataset [52], while VGG-16 is built on two image datasets: ImageNet-1K [53] and CIFAR-10 [54], with details listed in Table II. Batch sizes are configured to maximize the pipeline efficiency. We build a random forest [46] to evaluate the performance decline caused by dynamic computing environment.

3) Baselines: We introduce the following baselines for comparisons.

- *PipeDream-Flush [18]:* A synchronous pipelined training framework which follows 1F1B scheduling to improve the training efficiency.
- *PipeDream [11]:* The most notable asynchronous pipeline framework, which trains DNNs under 1F1B scheduling.
- *PipeDream-2BW [18]:* PipeDream-2BW is a memory-efficient variant of PipeDream, while sharing a similar asynchronous pipelined training pattern.
- *PipeDream-Re:* Pipeline via PipeDream while restarting the training for model re-partition at epoch boundary.
- *DynPipe-w/o RF:* a variant of DynPipe which uses stale interference information instead of random forest to evaluate performance decline caused by external task.

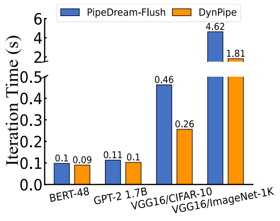
4) Parameters: For image classification tasks, we set the minibatch sizes and learning rates to 420, 32 and 0.01, 0.001, respectively. As for text processing, the minibatch size is 8. These parameter settings are chosen based on typical configurations that optimize training efficiency relative to the DNN model and GPU memory size.

5) Type of Interference: We incorporate the common interference into our evaluations [15], [44], including GPU sharing and spot instance interruptions. Specifically, we simulate interruptions in GPU sharing by introducing additional workloads on certain GPUs. To cover different interruption levels, we introduce a light interference with an external task queue at a specific arrival rate, and a heavy one where the task arrival rate is tripled. For the spot instance situation, we follow the conventions of Amazon AWS [55], where instances have a 10% chance of being preempted every five minutes.

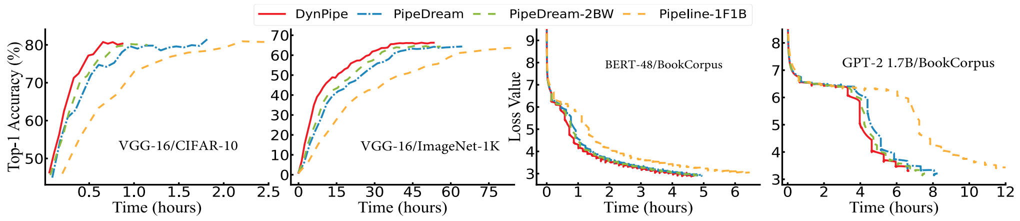
6) Metrics: We use two metrics to evaluate DynPipe and baselines. Completion time (CT) measures the time required to achieve convergence. Test accuracy denotes the model accuracy on test dataset. As for pre-training BERT-48 and

TABLE II
CT OF DYNPIPE AND BASELINES ON DIFFERENT DATASETS WITH SPEEDUP COMPARED TO PIPEDREAM-FLUSH

Task	Dataset	# Samples	Model	Loss/Final Accuracy	System	CT (h)	Speedup
Text Processing	BookCorpus	74004228	BERT-48	3.0	PipeDream-Flush	6.021	1.00×
					PipeDream	4.315	1.395×
					PipeDream-2BW	4.232	1.422×
					DynPipe	4.024	1.496×
			GPT-2 1.7B	3.5	PipeDream-Flush	11.473	1.00×
					PipeDream	6.59	1.741×
					PipeDream-2BW	6.32	1.815×
					DynPipe	5.95	1.928×
Image Classification	ImageNet-1K	1350000	VGG-16	0.6	PipeDream-Flush	62.69	1.00×
					PipeDream	37.816	1.658×
					PipeDream-2BW	32.82	1.91×
					DynPipe	27.859	2.25×
	CIFAR-10	60000	VGG-16	0.8	PipeDream-Flush	2.207	1.00×
					PipeDream	1.816	1.215×
					PipeDream-2BW	0.947	2.33×
					DynPipe	0.654	3.375×



(a) Ave. iteration time.



(b) Test accuracy and global loss

Fig. 11. Test accuracy/global loss over time, and iteration time of DynPipe and baselines on different datasets.

GPT-2 1.7B/12B, we employ global loss to replace test accuracy.

B. End-to-End Pipeline Planning

First, we use Cluster A to show the results in stable computing environments with negligible external interruptions.

1) CT Mitigation: Table II shows the model convergence time, or CT, of DynPipe and baselines, when BERT-48/GPT-2 1.7B and VGG-16 approximately reach converged loss or accuracy, respectively. DynPipe achieves convergence within a much shorter time, which is 1.07-2.78× faster than PipeDream, 1.05-1.46× than PipeDream-2BW, and 1.496-3.375× than PipeDream-Flush while maintaining a comparable test accuracy or global loss. This substantial improvement is attributed to the end-to-end optimization that accounts for non-overlapped communication time, facilitating an optimal model partition.

Furthermore, we also integrate DynPipe's approach into the synchronous pipeline of PipeDream-Flush, accounting for communication overhead. As depicted in Fig. 11(a), this end-to-end optimization can boost training speeds by 1.11-2.55× across various models, illustrating its versatility in synchronous training scenarios. Combined with Fig. 11(b) which illustrates the time-to-accuracy on the four datasets, DynPipe consistently outperforms benchmarks, reducing CT by 5-76% without compromising model performance. In summary, DynPipe can achieve faster convergence than benchmarks, while also preserving the model accuracy.

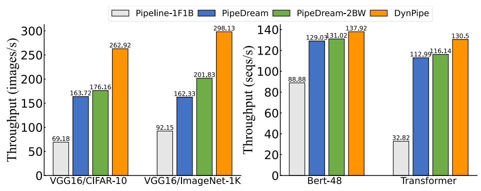


Fig. 12. Throughput for image and text tasks.

2) Training Throughput: The throughput of processing input data is also a critical metric. Fig. 12 draws the average training throughput under the same settings as Fig. 11(b). On image datasets, DynPipe improves the throughput by 60.6-83.7%, 47.7-49.2% and 280.1-423.5% compared to PipeDream, PipeDream-2BW and PipeDream-Flush, respectively. The improvements become 6.8-15.4%, 5.3-12.4% and 55.2-297.6% for BERT-48 and GPT-2 1.7B on text dataset.

3) Memory Consumption: Asynchronous pipelined training needs to maintain the model parameters from various iterations, leading to increased memory consumption. To address this issue, DynPipe offloads the intermediate parameters to CPU, thereby reducing GPU footprint. Fig. 13 shows the highest memory consumption in the first model stage, under the same conditions as Fig. 11(b). PipeDream exhibits the highest memory use on BERT-48 and GPT-2 1.7B due to its need to cache multiple copies of model parameters in GPU memory. In contrast, PipeDream-Flush caches only the activations of in-flight

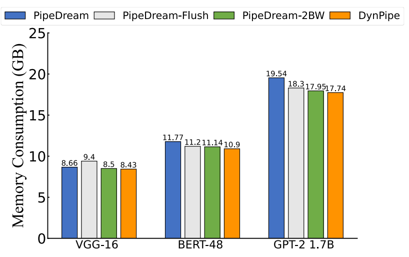


Fig. 13. Memory consumption.

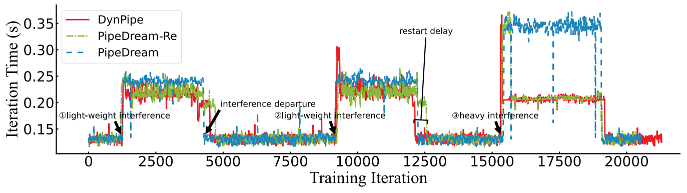


Fig. 14. Sensitivity of DynPipe and baselines.

minibatches, which occupy less memory than model parameters for BERT-48 and GPT-2 1.7B, but not for VGG-16. DynPipe is slightly better than PipeDream-2BW in any case, as it only saves one copy of the parameters in GPU. DynPipe, utilizing flexible offloading and swapping operations, consistently achieves superior memory efficiency with the lowest GPU footprint across all scenarios.

C. Adaptive Model Re-Partition

DynPipe can effectively handle the dynamic task interference. To verify this, we analyze task interference as described in Section VII-A5, by continuing training based on the steady pipeline plan.

1) Sensitivity to Interference: W.l.o.g., we train VGG-16 on CIFAR-10 using Cluster A to obtain per-iteration time when task interference is present. From Fig. 14, when external task minimally disrupts operations with negligible GPU SM consumption (① and ②), the advantage of DynPipe is subtle, offering a modest time reduction of 10% over PipeDream. However, with substantial workloads (③), time reduction escalates to 40%. PipeDream-Re needs to delay model re-partition for an epoch due to training restarts, making it respond more sluggishly to task interference than our DynPipe. Therefore, our DynPipe can well accommodate pipelined DNN training in dynamic computing environment.

2) Dynamic Re-Partition: Figs. 15 and 16 exhibit the time-to-accuracy and average iteration time on Cluster A, respectively, when encountering different levels of task interference. A significant performance downturn is noted, with iteration under PipeDream taking 1.2-3× longer time per update. DynPipe rapidly re-establishes load balance, cutting the time by 39-67% which equates to a 25-64% decrease in CT across the board given heavy task interference. For DynPipe-w/o RF, overall CT reduction is 10-41%. However, DynPipe-w/o RF yields inferior

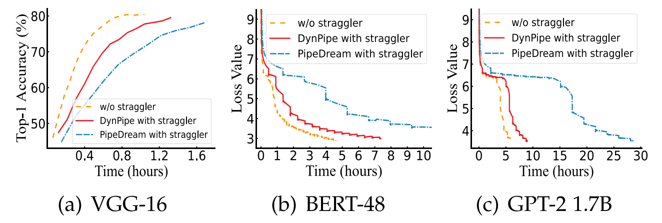


Fig. 15. Test accuracy/global loss under heavy interference.

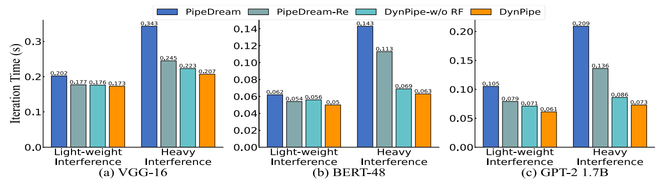


Fig. 16. Ave. iteration time under task interference.

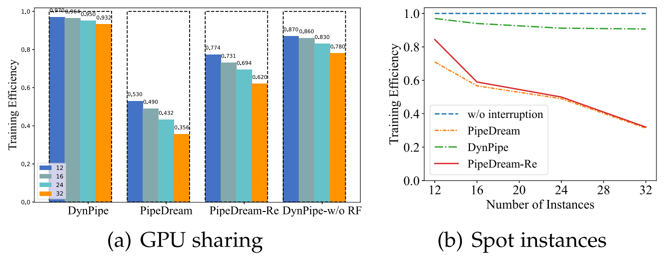


Fig. 17. Ave. iteration time.

performance than PipeDream-Re for BERT-48, which implies inaccurate stale interference information can mislead the workload migration process. Regarding PipeDream-Re, we witness a 10-33% CT reduction, showing DynPipe's ability to seamlessly redeploy pipelined training in dynamic environments.

We also evaluate performance by running GPT-2 12B on the larger Cluster B, considering both GPU sharing and spot instance interference, where the spot instance trace is extracted from [15]. We monitor the effective training time ratio, indicating the proportion of productive training time achieving convergence despite random stragglers. As depicted in Fig. 17, effective training time declines with larger GPU deployments due to increased interference likelihood, which often hampers training. Thanks to its GPU-level granularity adjustment, DynPipe maintains high effective training time of 90.7% and 93.2%, marking improvements of 8.7-188.9% and 19.4-161.8% over baseline systems. This superior performance stems from DynPipe's ability to quickly and transparently perform model re-partition and layer migration.

3) Ablation Study: We continue to perform ablation study to validate the transparent layer migration, using the training iterations required to complete re-partition as the metric.

Pipeline Checkpointing: Checkpointing system state aids in model re-partition given external interference. DynPipe minimizes the checkpointing time by pipelining the serialization

TABLE III
AVE. ITERATIONS REQUIRED TO COMPLETE RE-PARTITION

Flow Control	Asyn. Checkpoint	VGG	BERT-48	GPT-2 1.7B
x	x	174.4	382.9	466
x	✓	26	31.2	56.5
✓	x	132.3	254.1	302.6
✓	✓	9.8	15.9	35.1

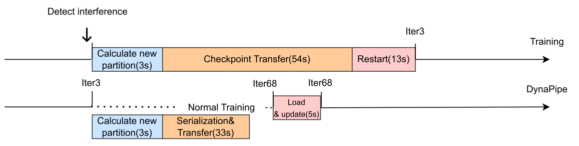


Fig. 18. Model re-partition of GPT-2 12B.

and distribution phases. As shown in Table III when using Cluster A, absence of pipeline leads to a 8-15× increase in re-partition iterations across various models, highlighting its role in maintaining DynPipe's efficiency.

Flow Control: Checkpoint distributions would compete with training traffic for limited network bandwidth. On Cluster A, Table III shows that the number of iterations taken per re-partition can increase by 61-165% without priority-based flow control, which is expected to worsen as more GPUs are utilized, positioning communication as the main bottleneck in pipelined training.

Re-partition overhead: DynPipe employs a transparent layer migration strategy that integrates model re-partition overhead into the training workflow, as shown in Fig. 18 with GPT-2 12B on Cluster B. This approach reduces reconfiguration time from 70 s (in restart-based methods) to just 5 s for model loading. When using a larger model, GPT-2 22B, along with GPT-2 12B, and simulating varying interference levels by injecting system disturbances every 10000, 5000, and 1000 iterations, DynPipe maintains up to 91.3%-98.3% training efficiency and achieves performance gains of 10.7%-198.9% over the baseline. Thus, our transparent layer migration proves effective in dynamic environments.

VIII. Related Work

Distributed DNN Training: With increasing DNN model size, training DNNs on a single machine is inefficient due to limited processing speed and memory capacity. Distributed DNN enables partitioning the datasets or model across distinct GPUs for parallel processing, known as data parallelism [56], [57] or model parallelism [58]. To enhance training efficiency, asynchronous pipeline parallelism is proposed to allow for a pipelined computation and communication [59], yet it introduces stage-wise model staleness. Existing pipeline systems mainly focus on optimizing the hardware speed, overlooking the staleness impact on statistical efficiency and oversimplifying pipeline formulation by removing inter-stage communication time. Both will compromise end-to-end performance.

Dynamic Training Scheduling: Computing environments are subject to fluctuations due to incoming tasks competing for resources [32]. Geng et al. propose ElasticPipe to tune workloads across workers when there are stragglers [13], [60]. vPipe proposes an adaptive approach for redistributing DNN layer partitions across GPUs, aiming to find the most efficient partition configuration through iterative stage re-balancing [12]. Although these methods are effective, they lack the ability to provide intra-stage parallelization and fail to handle external task interference.

Task Interference: Machine learning tasks are the predominant workload in modern cloud data centers, sparking significant interest in their efficient scheduling [61]. To improve GPU utilization, allocating multiple tasks to a single GPU is viable when the execution interference is minimal [32]. However, existing works mostly focus on task allocations in GPU clusters. When it comes to distributed DNN training, a semi-dynamic load balancing is developed to address GPU resource scarcity, yet it mainly explores data parallelism situation [62]. Checkpointing is crucial for managing the volatile hardware environments resulting from task interference. CheckFreq [21] enhances checkpointing efficiency by decoupling the checkpointing process, although its performance is constrained by the bandwidth of remote storage and variable straggler patterns. In contrast, Gemini [22] facilitates swift error recovery through in-memory checkpointing but does not support popular parallelism strategies like tensor and pipeline parallelism. Meanwhile, DLRover-RM [63] accommodates various parallelism methods but functions at the job level rather than the GPU level, which disrupts the continuous training flow with its checkpoint distribution.

IX. Conclusion

In this paper, we propose DynPipe to enhance the end-to-end training performance in dynamic computing environments. DynPipe can strike an optimal balance between the hardware speed and statistical efficiency by incorporating a staleness-aware training convergence and accurate pipeline profiling. Considering external interference, DynPipe deploys a non-intrusive random forest model that utilizes runtime stage statistics to pinpoint potential stragglers. On this basis, DynPipe restores both intra and inter-stage load balancing, thus mitigating the adverse impact of unexpected interruption. Extensive experiments on real-world datasets show that DynPipe can expedite the time-to-accuracy over benchmarks by at least 1.5-3.4×.

References

[1] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Assoc. Comput. Linguistics Hum. Lang. Technol.*, 2019, pp. 4171–4186.
[2] A. Vaswani et al., "Attention is all you need," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
[3] A. Radford, "Improving language understanding by generative pre-training," 2018.
[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. Int. Conf. Learn. Representations*, 2015.
[5] A. Meta, "Introducing meta Llama 3: The most capable openly available LLM to date," *Meta AI*, 2024.
[6] Z. Lin et al., "nnScaler: Constraint-guided parallelization plan generation for deep learning training," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2024, pp. 347–363.

[7] X. Miao et al., “Galvatron: Efficient transformer training over multiple GPUs using automatic parallelism,” in *Proc. VLDB Endowment*, vol. 16, no. 3, pp. 470–479, 2022.

[8] X. Miao, Y. Shi, Z. Yang, B. Cui, and Z. Jia, “SDPipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training,” in *Proc. VLDB Endowment*, vol. 16, no. 9, pp. 2354–2363, 2023.

[9] Y. Huang et al., “GPipe: Efficient training of giant neural networks using pipeline parallelism,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 103–112.

[10] S. Fan et al., “DAPPLE: A pipelined data parallel approach for training large models,” in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 431–445.

[11] D. Narayanan et al., “PipeDream: Generalized pipeline parallelism for DNN training,” in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 1–15.

[12] S. Zhao et al., “vPipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel DNN training,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 3, pp. 489–506, Mar. 2022.

[13] X. Miao, X. Nie, H. Zhang, T. Zhao, and B. Cui, “Hetu: A highly efficient automatic parallel distributed deep learning system,” *Sci. China Inf. Sci.*, vol. 66, no. 1, 2023, Art. no. 117101.

[14] Q. Weng et al., “Beware of fragmentation: Scheduling GPU-sharing workloads with fragmentation gradient descent,” in *Proc. USENIX Annu. Tech. Conf.*, 2023, pp. 995–1008.

[15] X. Miao et al., “SpotServe: Serving generative large language models on preemptible instances,” in *Proc. ACM Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2024, pp. 1112–1127.

[16] J. Thorpe et al., “Bamboo: Making preemptible instances resilient for affordable training of large DNNs,” in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2023, pp. 497–513.

[17] H. Li et al., “Malleus: Straggler-resilient hybrid parallel training of large-scale models via malleable data and model parallelization,” in *Proc. ACM Manag. Data*, vol. 3, no. 3, pp. 185:1–185:28, 2025.

[18] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, “Memory-efficient pipeline-parallel DNN training,” in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 7937–7947.

[19] Y. Zhao, Y. Liu, Y. Peng, Y. Zhu, X. Liu, and X. Jin, “Multi-resource interleaving for deep learning training,” in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 428–440.

[20] A. Jajoo, Y. C. Hu, X. Lin, and N. Deng, “A case for task sampling based learning for cluster job scheduling,” in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 19–33.

[21] J. Mohan, A. Phanishayee, and V. Chidambaram, “CheckFreq: Frequent, fine-grained DNN checkpointing,” in *Proc. USENIX Conf. File Storage Technol.*, 2021, pp. 203–216.

[22] Z. Wang et al., “GEMINI: Fast failure recovery in distributed training with in-memory checkpoints,” in *Proc. Symp. Operating Syst. Princ.*, 2023, pp. 364–381.

[23] M. Isaev, N. McDonald, L. Dennison, and R. W. Vuduc, “Calculon: A methodology and tool for high-level co-design of systems and large language models,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2023, pp. 71:1–71:14.

[24] K. Ge, K. Lu, Y. Fu, X. Deng, Z. Lai, and D. Li, “Compressed collective sparse-sketch for distributed data-parallel training of deep learning models,” *IEEE J. Sel. Areas Commun.*, vol. 41, no. 4, pp. 941–963, Apr. 2023.

[25] Y. Peng et al., “A generic communication scheduler for distributed DNN training acceleration,” in *Proc. ACM Symp. Operating Syst. Princ.*, 2019, pp. 16–29.

[26] H. Touvron et al., “Llama 2: Open foundation and fine-tuned chat models,” 2023, *arXiv:2307.09288*.

[27] OpenAI, “GPT-4 technical report,” 2023, *arXiv:2303.08774*.

[28] J. Dean et al., “Large scale distributed deep networks,” in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1232–1240.

[29] Single-machine model parallel best practices, 2024. [Online]. Available: https://pytorch.org/tutorials/intermediate/model_parallel_tutorial.html

[30] S. Zhao et al., “NASPipe: High performance and reproducible pipeline parallel supernet training via causal synchronous parallelism,” in *Proc. ACM Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2022, pp. 374–387.

[31] E. Shi et al., “Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond,” in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 39–51.

[32] Q. Hu, M. Zhang, P. Sun, Y. Wen, and T. Zhang, “Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs,” in *Proc. ACM Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2023, pp. 457–472.

[33] X. Liu et al., “MuxFlow: Efficient GPU sharing in production-level clusters with more than 10000 GPUs,” *Sci. China Inf. Sci.*, vol. 67, no. 12, 2024, Art. no. 222101.

[34] Q. Weng et al., “MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters,” in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 945–960.

[35] Z. Jiang et al., “MegaScale: Scaling large language model training to more than 10, 000 GPUs,” in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2024, pp. 745–760.

[36] D. Narayanan et al., “Efficient large-scale language model training on GPU clusters using megatron-LM,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, Art. no. 58.

[37] L. Zheng et al., “Alpa: Automating inter- and intra-operator parallelism for distributed deep learning,” in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2022, pp. 559–578.

[38] J. H. Park et al., “HetPipe: Enabling large DNN training on (Whimpy) heterogeneous GPU clusters through integration of pipelined model parallelism and data parallelism,” in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 307–321.

[39] H. Zhang, Y. Tang, A. Khandelwal, and I. Stoica, “SHEPHERD: Serving DNNs in the wild,” in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2023, pp. 787–808.

[40] NVIDIA Nsight systems, 2024. [Online]. Available: https://developer.nvidia.com/nsight-systems

[41] B. Luo, W. Xiao, S. Wang, J. Huang, and L. Tassiulas, “Tackling system and statistical heterogeneity for federated learning with adaptive client sampling,” in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 1739–1748.

[42] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.

[43] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, “Analysis of large-scale multi-tenant GPU clusters for DNN training workloads,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 947–960.

[44] W. Chen, Z. Mo, H. Xu, K. Ye, and C. Xu, “Interference-aware multiplexing for deep learning in GPU clusters: A middleware approach,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2023, pp. 30:1–30:15.

[45] G. Yeung, D. Borowiec, R. Yang, A. Friday, R. Harper, and P. Garraghan, “Horus: Interference-aware and prediction-based scheduling in deep learning systems,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 88–100, Jan. 2022.

[46] L. Breiman, “Random forests,” *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[47] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, “Regularized evolution for image classifier architecture search,” in *Proc. AAAI Conf. Artif. Intell.*, 2019, pp. 4780–4789.

[48] J. F. Shortle, J. M. Thompson, D. Gross, and C. M. Harris, *Fundamentals of Queueing Theory*. Hoboken, NJ, USA: Wiley, 2018.

[49] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, “DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters,” in *Proc. 26th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2020, pp. 3505–3506.

[50] PyTorch, 2024. [Online]. Available: https://pytorch.org/

[51] A. Radford et al., “Language models are unsupervised multitask learners,” *OpenAI Blog*, vol. 1, no. 8, 2019, Art. no. 9.

[52] Y. Zhu et al., “Aligning books and movies: Towards story-like visual explanations by watching movies and reading books,” in *Proc. IEEE Int. Conf. Comput. Vis.*, 2015, pp. 19–27.

[53] ImageNet, 2021. [Online]. Available: https://www.image-net.org/

[54] A. Krizhevsky et al., “Learning multiple layers of features from tiny images,” 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:18268744

[55] Amazon EC2 spot instance, 2024. [Online]. Available: https://aws.amazon.com/cn/ec2/spot/

[56] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, “A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters,” in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2020, pp. 463–479.

[57] Z. Lai et al., “Merak: An efficient distributed DNN training framework with automated 3D parallelism for giant foundation models,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 5, pp. 1466–1478, May 2023.

[58] S. Li, K. Lu, Z. Lai, W. Liu, K. Ge, and D. S. Li, "A multidimensional communication scheduling method for hybrid parallel DNN training," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 8, pp. 1415–1428, Aug. 2024.
[59] S. Eliad, I. Hakimi, A. D. Jagger, M. Silberstein, and A. Schuster, "Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 381–396.
[60] J. Geng, D. Li, and S. Wang, "ElasticPipe: An efficient and dynamic model-parallel solution to DNN training," in *Proc. 10th Workshop Sci. Cloud Comput.*, 2019, pp. 5–9.
[61] W. Xiao et al., "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 595–610.
[62] C. Chen, Q. Weng, W. Wang, B. Li, and B. Li, "Semi-dynamic load balancing: Efficient distributed learning in non-dedicated environments," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 431–446.
[63] Q. Wang et al., "DLRover-RM: Resource optimization for deep recommendation models training in the cloud," in *Proc. VLDB Endowment*, vol. 17, no. 12, pp. 4130–4144, 2024.

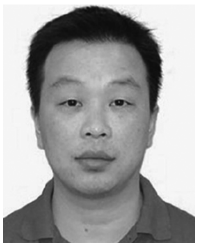
Zhengyi Yuan received the BS degree from the School of Central South University, Hunan, China, in 2023. He is currently working toward the PhD degree with the School of Huazhong University of Science and Technology, Wuhan, China. His current research interests include distributed machine learning system, and large language model inference system.

Xiong Wang (Member, IEEE) received the BE degree in electronic information engineering from the Huazhong University of Science and Technology, Wuhan, China, in 2014, and the PhD degree in electronic engineering from Shanghai Jiao Tong University, Shanghai, China, in 2019. He was a post-doctoral fellow with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong, China, from 2019 to 2021. He is currently an associate professor with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. He is a member of the ACM. His research interests include distributed machine learning systems, federated learning, and data-center network.

Yuntao Nie received the BS degree from the School of Central South University, Hunan, China, in 2023. He is currently working toward the MS degree with the School of Huazhong University of Science and Technology, Wuhan, China. His current research interests include distributed machine learning system, and large language model inference system.

Yufei Tao received the BS degree from the School of Huazhong University of Science and Technology, Wuhan, China, in 2023. He is currently working toward the MS degree with the School of Huazhong University of Science and Technology, Wuhan, China. His current research interests include distributed machine learning system, and large language model inference system.

Yuqing Li (Member, IEEE) received the BS degree in communication engineering from Xidian University, Xi'an, China, in 2014, and the PhD degree in electronic engineering from Shanghai Jiao Tong University, Shanghai, China, in 2019. She is currently an associate professor with the School of Cyber Science and Engineering, Wuhan University, China. Before this, she was a researcher with Huawei Hong Kong Research Center from 2020–2022, and a postdoctoral fellow with the Hong Kong University of Science and Technology from 2019–2020. Her research interests include the area of data-intensive and machine learning systems, cloud and edge computing, data privacy, and security.


Zhiyuan Shao (Member, IEEE) received the PhD degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2005. He is now a professor with the National Engineering Research Center for Big Data Technology and System, HUST. His research interests include the areas of graph computing, big-data processing, and computing systems.

Xiaofei Liao (Member, IEEE) received the PhD degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2005. He is currently a professor with the School of Computer Science and Technology, HUST. His research interests include computer architecture, system software, and Big Data processing. He was the recipient of the Excellent Youth Award from the National Science Foundation of China in 2018 and the CCF-IEEE CS Young Computer Scientist Award in 2017. He is a member of the IEEE Computer Society and ACM.

Bo Li (Fellow, IEEE) received the BEng (summa cum laude) degree in computer science from Tsinghua University, Beijing, and the PhD degree in electrical and computer engineering from the University of Massachusetts at Amherst. He is a chair professor with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology. He held a Cheung Kong visiting chair professor with Shanghai Jiao Tong University between 2010 and 2016, and was the chief technical advisor for ChinaCache Corp. (NASDAQ:CCIH), a leading CDN provider. He was an adjunct researcher with the Microsoft Research Asia (MSRA) (1999–2006) and with the Microsoft Advanced Technology Center (2007–2008). He made pioneering contributions in multimedia communications and the Internet video broadcast, in particular Coolstreaming system, which was credited as first large scale Peer-to-Peer live video streaming system in the world. It attracted significant attention from both industry and academia and received the Test-of-Time Best Paper Award from IEEE INFOCOM (2015). He has been an editor or a guest editor for more than two dozen of IEEE and ACM journals and magazines. He was the Co-TPC chair for IEEE INFOCOM 2004.

Hai Jin (Fellow, IEEE) received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST), China, in 1994. He is a chair professor of computer science and engineering with the Huazhong University of Science and Technology, China. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. He worked with the University of Hong Kong between 1998 and 2000, and as a visiting scholar with the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. He is a fellow of the CCF, and a life member of the ACM. He has co-authored more than 20 books and published more than 900 research papers. His research interests include computer architecture, parallel and distributed computing, Big Data processing, data storage, and system security.