PipeChurn: A Reliable Pipeline Parallelism Framework using Erasure Coding

Kai-Siang Wang, Mayank Bhatia, Mingyue Tang
University of Illinois, Urbana Champaign
{kw37,mayankb3,mt55}@illinois.edu

Abstract

Training large deep learning models today relies on different parallelism techniques to speed up the time to convergence. With the emergence of edge devices and their computation capability, previous work has adapted the training jobs to an edge environment. Combined with the churns, a normal phenomenon in such a setting, the long-running jobs make reliability especially important when building a system. In this paper, we introduce PipeChurn, a reliable pipeline parallelism framework using Erasure Coding, with the following properties: highly fault-tolerant, fast recovery, and reconfigurability. The simulation results show that PipeChurn can reduce the end-to-end latency by 25.4% with simultaneous failures and have a 3.1x smaller recovery delay from failures. The real deployment also shows that PipeChurn can improve end-to-end latency by up to 10%.

1 Introduction

Large Models Deep Learning Training, the rise of generative foundation models, including Large Language Models (LLMs), Large Vision Models (LVMs), and Large Multi-Modal Models (LMMs), has shown promising results across both academic and industrial fields [5, 30, 36]. However, training these large models demands significant time and computational resources. For instance, training LLaMA, an open-source LLM with 65 billion parameters, demands 2048 A100 GPUs, each equipped with 80GB of RAM, over a span of 21 days [26]. To optimize the training efficiency of such models, Deep Learning Training (DLT) jobs have become standard practice, involving Python scripts defining the model and its training procedures. Each iteration of a DLT job processes mini-batches of training data, applies loss functions during forward passes, computes gradients in backward passes for each parameter, and updates model parameters via gradient descent.

Basic methods for training large models include data parallelism [13,22] and model parallelism [23]. Data parallelism partitions data across multiple nodes with a local copy of the model weights. However, data parallelism introduces communication overheads proportional to the number of model weights as all workers exchange model updates (e.g., All-Reduce [13]). In model parallelism, operators are partitioned across nodes, and each worker is responsible for a subset of the model parameters. However, model parallelism alone is often insufficient to expedite the training of large models [31]. Recent work demonstrates that pipeline parellelism achieves low communication overhead and high resource utilization, effectively combating inefficiencies in systems training large models [17].

The environment on which efficient parallelism techniques are implemented impacts performance and privacy. Traditional cloud-based deep learning solutions offer strong computing power. However, they may lead to significant delays due to communication overheads as well as data privacy and security issues [9]. Edge computing may alleviate these issues with improved user experience, especially as strong computing power becomes increasingly available in edge nodes closer to the user side. Utilization of edge nodes for deep learning training in the future is supported by Gartner's estimation that around 75% of all enterprises will generate and process data outside of a centralized data center or cloud by 2025 [1].

Existing work doesn't consider new arrivals in pipeline parallelism. Edge environments are typically more dynamic than data centers. If an edge server handling specific tasks or storing specific data goes offline unexpectedly, mechanisms must exist for failover without disruption. If a worker drops out, incremental repartitioning protects prior work and prevents shutdowns. Similarly, it is valuable to incorporate any new workers that would otherwise go unused. Previous work describes recovery modes with repartitioning and weight redistribution for training on edge devices, but these shutdowns unnecessarily halt progress [6, 25, 27].

Due to privacy and communication cost concerns, alongside the growing prevalence of large model training, there is a need for a reliable and efficient training framework with low communication overheads and minimal security concerns. Middle-edge computing layers such as mist computing and fog computing have recently gained more traction for these reasons [15, 33]. This infrastructure connects end-edge devices to central servers and can act as a helpful middle ground with some of the privacy benefits of an end-edge device alongside greater computation capabilities. However, high node churn is notable in such environments due to various factors such as device failure and network connectivity [33]. Thus, existing fault-tolerant solutions to pipeline parallelism, such as FTPipeHD which has a central node for managing failures [6], are ill-equipped for these environments. PipeChurn decentralizes the recovery of pipeline parallelism to ensure reliable training in middle-edge environments, allowing for an arbitrary number of worker failures. Furthermore, PipeChurn quickly recovers from failures by only recomputing a necessary fraction of activations after a failure.

2 Background & Motivation

In this section, we discuss pipeline parallelism and its usefulness in middle-edge environments.

2.1 Pipeline Parallelism

Pipelining is a parallelization dimension for DNN training. This optimization technique achieves low communication overhead and high resource utilization for certain types of models, often improving training performance with no reduction in accuracy [10].

In pipeline parallelism, models are divided into networks of stages [17, 32]. Each stage carries out operations using only values received from adjacent stages as well as its own internal state. Each device in the pipeline is assigned to a number of stages. A given stage only needs to communicate with stages immediately before and after itself in the pipeline, enabling a communication-efficient parallelization scheme [8]. The granularity of the stages determines the ratio of computation to communication. A fine-grained stage will carry out only a few operations and communicate extremely frequently.

In standard pipeline parallelism, a batch of training samples is sharded into smaller microbatches, and workers process different microbatches concurrently. This concurrency allows for training speedups for large models because stages without dependencies can run ahead of stages waiting for input. In other words, different workers can work on different microbatches simultaneously [10].

Implementing pipeline parallelism for distributed model training is difficult because the training is both bidirectional and stateful [17]. For training, a forward pass through the model is generally followed by a backward pass using the same set of samples, which updates weight parameters. Intermediate outputs and weight parameters used in the forward pass are also required in the backward pass. Naïve pipelining is vulnerable to weight version mismatches across the forward and backward passes that significantly reduce the accuracy of the final trained model. PipeDream versions intermediate state data to promise clean weight update semantics. Various methods of stashing weight versions are employed to trade-off between memory footprint, throughput, and the number of samples used before each weight update [17]. Furthermore, PipeDream automatically optimizes the partition of operators across workers by reasoning about communication requirements and computational capabilities of different devices.

Pipeline parallelism can also be composed with other parallelization strategies like tensor and ZeRO-Style data parallelism [20]. Exploiting these techniques together (3D parallelism) enables training models with over a trillion parameters.

2.2 Fault Tolerance in Middle Edge Environments & Churn

Replication, decentralization, and load balancing are key techniques used to strengthen fault tolerance in edge environments [18, 34].

Checkpoint-restart is a common fault tolerance technique in which the training state is saved at a certain point. In the case of failure, training is restored from the last valid checkpoint [21]. However, in edge environments, we cannot checkpoint on the disk of a single node because that node can fail. An effective and efficient replication strategy for edge environments is crucial to fault tolerance but non-trivial.

Existing works like FTPipeHD have a central node to recover system failures with weight redistribution [6]. However, this fault tolerance mechanism is still susceptible to a single point of failure. PipeChurn decentralizes recovery to ensure reliable pipeline parallelism in unreliable edge deployments.

Bamboo and Gemini are recent systems which seek to increase fault tolerance for DNN training [25, 28]. Bamboo increases fault tolerance by performing redundant computations during pipeline bubbles [25]. Bubbles represent inefficiency where a worker is idle because they are dependant on receiving some other computation before contributing to the pipeline. A bubble is shown in the right side of figure 1. Bamboo enables fast recovery through in-bubble redundant computations, but it can only tolerate one failure at a time. Similarly, Gemini supports fast failure recovery by exploiting the high bandwidth of CPU memory, but it cannot tolerate two failures in the same group [28]. Through erasure coding, PipeChurn tolerates an arbitrary number of worker failures.

High node churn is notable in edge environments due to volatile, unreliable, and unpredictable volunteer resources. Middle edge paradigms such as mist and fog computing act as a waystation between edge and cloud computing, in which infrastructure connects end devices with the central server [15,33]. As DNN training becomes more prevalent, it is costly communication and low privacy to send all raw data to the cloud. Mist and fog computing layers are becoming more powerful and have improved privacy compared to traditional cloud servers, but they still suffer from churn [33]. Churn occurs due to various factors, such as device failures, network connectivity issues, and deliberate reconfiguration. Thus, strategies to mitigate churn in fog computing environments are critical middle-edge applications.

3 PipeChurn Design

In this section, we describe the design of *PipeChurn*, highlighting how it provides high fault tolerance and fast recovery.

3.1 High-level architecture

PipeChurn classifies nodes into two types: **worker nodes** and **memory nodes**, depicted in Figure 1. This classification is based on their computational ability and memory capacity. Essentially, a worker node possesses greater computing power but less

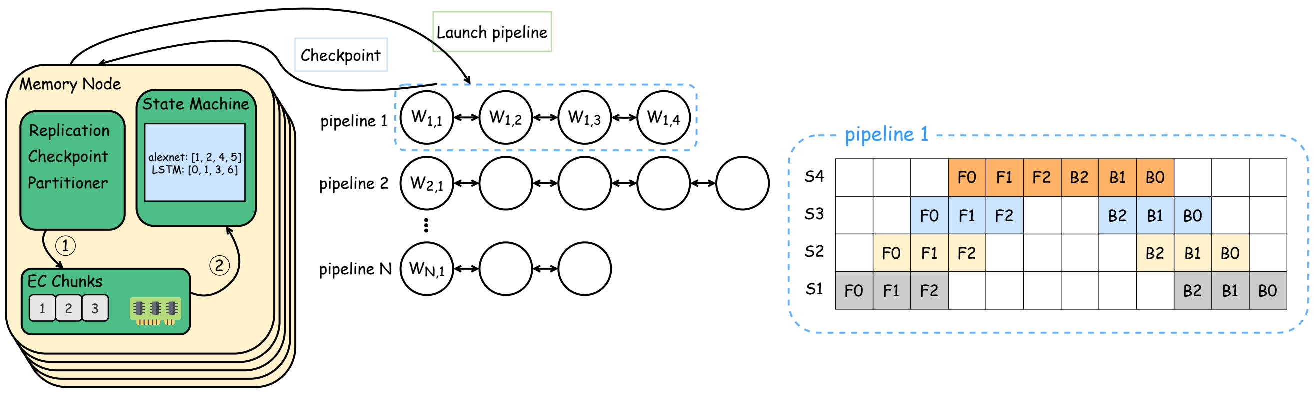


Figure 1: Illustration of the PipeChurn system architecture. Arrow 1 represents the flow of how the memory node updates erasure-coded chunks due to replication and checkpointing. Arrow 2 represents the flow of how the EC chunk information is maintained through the SMR system.

memory, whereas a memory node boasts a larger memory but is constrained in computing power.

PipeChurn enables training multiple pipelines concurrently. The model of each pipeline is partitioned into P partitions (stages), and each partition is assigned to a worker $W_{i,j}$, representing pipeline i and stage j. In the same pipeline, *PipeChurn* uses *pipeline parallelism*, where worker $W_{i,j}$ gets input activations from the previous worker $W_{i,j-1}$, performs the forward pass computation, and sends the output activations to the next worker $W_{i,j+1}$. Similarly, worker $W_{i,j}$ receives gradients from worker $W_{i,j+1}$, performs the backward pass computation, and sends input gradients to worker $W_{i,j-1}$.

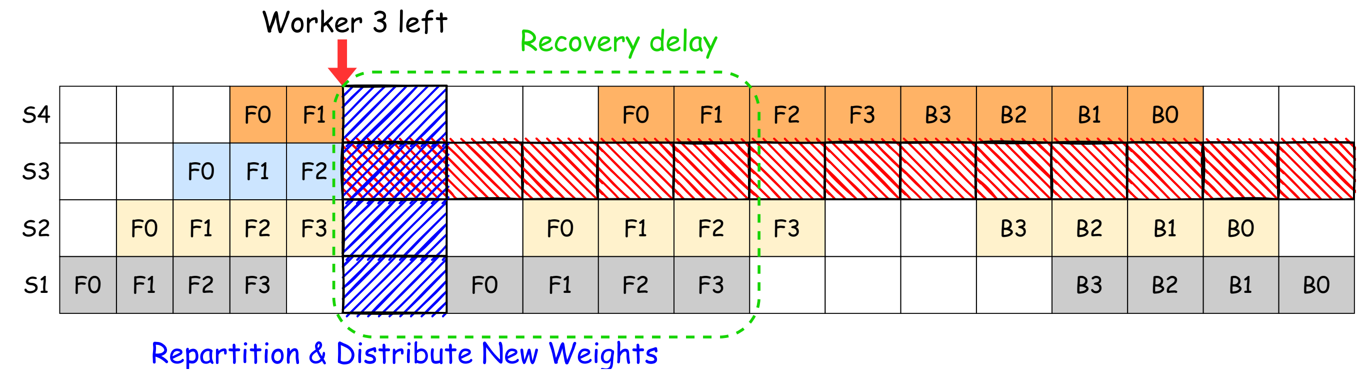
The model weights are divided into several data chunks and stored in-memory on memory nodes represented by the gray blocks. Among all memory nodes, a single leader is designated at any given time. This leader manages user requests, partitions the model according to specifications (e.g., the number of workers), and initiates the pipeline. The detailed model profiling and partitioning procedures will be explained in the next section.

3.2 Model Profiling and Partitioning

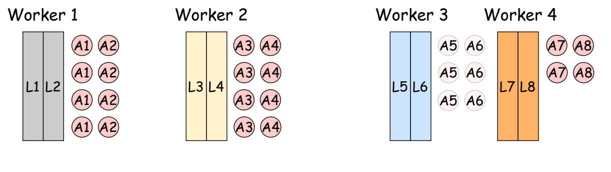
Following the design of PipeDream [17], *PipeChurn* optimizes distributed model training by outputting a balanced pipeline. Specifically, it segments each stage of the training into approximately equal blocks by the running time of each layer, distributing these among the workers. This division ensures that each block has a similar amount of computation time within a specific memory constraint. Since Deep Neural Network (DNN) training typically exhibits minimal variance in runtime across different inputs, *PipeChurn* profiles the model's running time based on the inputs' size and the duration of forward and backward passes. It logs the computation time and memory usage for each layer, utilizing these profiles as inputs to the partitioning algorithm. The partitioning strategy divides the model, prioritizing computation time while also considering memory usage, the number of workers, bandwidth, and other relevant specifications.

Profiler. *PipeChurn* current profiling deep learning models by recording two metrics for each layer: 1) forward pass time (T) and 2) memory usage (Mem). For each model, we have a short profiling run with 300 mini-batch and average the logged computation time (T^i) and memory (Mem^i) results for each layer i as input for the Partition algorithm.

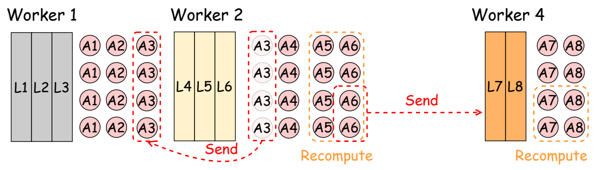
Partitioner. the *PipeChurn* model partitioner is designed to partition computational tasks among multiple workers in a way that minimizes the time taken by the slowest worker. The current implementation takes the average computation time of each layer (T^i), and uses dynamic programming to compute the optimal task partitioning (group of model layers



(a) A pipeline of a minibatch with 4 workers. F denotes forward pass, B denotes backward pass. The minibatch is restarted after the churn.



(b) Before Worker 3 leaves, the partition is (2, 2, 2, 2). Each worker keeps the activations for backward propagation. When Worker 3 leaves, activations 5 and 6 for the first three microbatches are lost.



(c) After Worker 3 leaves, the new partition is (3, 3, 2). Instead of recomputing all the microbatches, *PipeChurn* identifies the missing activations and only recomputes them.

Figure 2: *PipeChurn*'s fast recovery

$Model_{k,j}$, denotes as stage k) for each worker. The algorithm starts by preparing a cache or storage to remember the results of previous calculations. It contains a recursive function that calculates the minimum time for the slowest worker by exploring different ways to split the tasks using dynamic programming. The outcome of the algorithm returns the index k, j of layer groups ($Model_{k,j}$) according to the way of splitting different stages. A model partitioner will divide the model based on the returned layer indexes and send the sub-models to other workers.

3.3 Fault Tolerance

All of the memory nodes function as a State Machine Replication (SMR) system based on a leader-based consensus algorithm. Models are segmented into layers, and each layer is replicated on memory nodes using Erasure Coding (EC). The parameters of EC, denoted as (k, r), represent that the original data is partitioned into k data chunks and r additional parity chunks. This setup enables the system to tolerate up to r memory node failures and an arbitrary number of worker failures. The main benefit of using erasure coding is its efficiency in terms of size. For example, to accommodate four simultaneous memory node failures, using replication necessitates a minimum of five replicas, resulting in a fivefold increase in data size overhead. Conversely, utilizing erasure coding with parameters $(10, 4)$ incurs only a 1.4x data size overhead. The catch is the computational overhead. With a larger parameter setting, the encoding and decoding time of the same data will also increase. However, we will show in the experiments that this overhead is negligible during the synchronization.

When a memory node fails, the system needs to reconstruct the missing erasure-coded data. We let the memory node that has the smallest data chunk id to do the reconstruction. For example, if the first chunk of the data is lost, the memory node that has the second chunk takes the reconstruction task. In the case a worker node fails, we need to get a new partition for the pipeline and send the missing weights to each worker. *PipeChurn* proposes a fast recovery mechanism, described in section 3.5.

3.4 Checkpointing with EC

In a distributed training environment, checkpointing does not happen on just one node. In FTPipeHD, they assume there is a central node that will never fail, and all the model weights are backed up periodically on the central node. However, not only is the assumption weak in real-world situations, but the central node design limits the opportunity for better scalability. In our design, model weights are encoded into multiple chunks and replicated on multiple memory nodes. During the checkpointing, for each layer, we assign the memory node that has the first data chunk of such layer to update the corresponding data chunks and parity chunks. The checkpointing scales as the number of memory nodes increases since each EC update is independent. There are some works [24, 29] studying how to update erasure-coded data efficiently, and we believe their approaches are orthogonal to our approach.

3.5 Fast recovery

As a churn occurs, in the previous work, the model training has to be paused until the models are repartitioned and the corresponding weights are redistributed, as shown in Figure 2a. We define the recovery delay D as the time needed from the churn to the pipeline catches up to the same step. For example, in the figure, the pipeline has finished the forward passes of the first two microbatches before worker 3 leaves. The recovery delay, in this case, is the time F1 is recomputed minus the time worker 3 left, depicted as green dashed boxes.

Repartition. As the stage partitioning algorithm may take several seconds to generate a partition plan based on the number of layers and machines, we require an efficient method to repartition the model when a node exits or joins. Due to the nature of dynamic programming, a decrease in the number of nodes necessitates only a few recalculations, as most states are already computed. For example, when a user sets the number of workers as 8 when launching a pipeline, the system not only computes the partition with the 8 workers but also asynchronously computes the partition with 16 workers. This process mirrors how the capacity of a container changes in the STL, doubling the capacity when the size is close to it.

Seamless Recovery. When the churn happens, even though we cannot run the pipeline with the new partition before it is calculated and the weights are redistributed, it does not imply we have to pause the original pipeline. As shown in Figure 2c, we allow regular workers to continue executing forward and backward passes with their existing weights and keep all the activations and gradients. Given the rank of the lost worker, the server reconfigures the pipeline in the steps as shown in Algorithm 1. First, it calculates the new partition based on the current worker list and sends the missing weights based on the new partition. Second, from Lines 5-8, it calculates the missing layers for each partition. Specifically, it compares the layer difference of the new partition and the old partition. If a layer is in the new partition but not in the old partition, it is marked as "missing."

Upon receiving the new information, the worker updates the world, as shown in lines 10 to 16. The worker initially examines for any absent activations and gradients for each missing layer. If the missing activations and gradients are located on the failed worker, recomputation proceeds. Alternatively, the worker can simply request the missing elements from the other workers.

4 Implementation

In this section, we describe the detailed implementation of *PipeChurn*. Our current implementation

Algorithm 1 Fast Recovery

```
Input: $left\_rank, old\_partition$
 1: procedure CONFIGURE(on server)
 2:     1. get new partition
 3:     2. send new weights to workers
 4:     3. find the lost layers
 5:     for each partition(old,new) do
 6:         4. calculate the missing
    layers
 7:         5. send information to worker
 8:     end for
 9: end procedure
10: procedure UPDATE_WORLD(on worker)
11:     for each missing layer do
12:         1. recompute if lost
13:         2. request if on other worker
14:         3. resume the training
15:     end for
16: end procedure
```

of pipeline runtime is built on, but not limited to, the PyTorch framework.

Memory Node. We implemented the memory node in Golang with ZeroMQ based on the gossip-based membership list protocol [2]. All the information, including chunk locations and pipeline statuses, are shared among all memory through the Raft algorithm [19]. The memory nodes also maintain the statuses of the worker nodes (e.g., current loads, idleness, and churn history), and based on the information, the memory nodes can select a better worker set for a pipeline.

Worker Node. The worker nodes are also implemented in Golang with ZeroMQ and are part of the full membership list. The communication between the pipeline worker (written in Python) and the worker is through ZeroMQ. Two pipeline workers in the same pipeline communicate with each other directly through a p2p library and do not need to relay the message to the Golang process.

Partitioner and Scheduler. There is a well-known Dynamic Programming(DP) algorithm for finding the optimal partition given a model, which has been adopted by several work [3, 6, 7, 10–12, 17, 35]. We implemented our partitioner based on the algorithm and made it efficient for recomputation. For scheduling the stages on the workers, currently we implement the policy from GPipe. However, our solution can be extended to different scheduling policies.

Simulator. We implemented a PipeChurn simulator to verify our design. To have a more accurate simulator, we have conducted exhaustive profiling tasks to obtain information such as stage execution time, communication times of different data sizes, and erasure coding and decoding time. We implemented the baseline algorithm and our approach on top of this simulator.

5 Evaluation

In this section, we first compare the end-to-end latency of *PipeChurn* against prior systems. Second, we highlight how *PipeChurn* can recover quickly from frequent churns. Finally, we show the synchronization overhead of using *PipeChurn* and justify that it is negligible.

Experimental setup. We run the experiments based on the simulation and the real systems. For our workloads, we run our experiments with three different models: **Alexnet**, **Resnet101**, and **BERT**. We compare our approach with FTPipeHD and FTPipeHD+EC, where all the weights are backed up on memory nodes instead of the central node.

5.1 End-to-end training latency

We first evaluate the end-to-end training latency with different rates of churns. Figure 3 shows the result of training **Alexnet** of 1 epoch. Initially, we run the pipeline with 8 workers and later periodically fail the node by every two seconds. We can see that *PipeChurn* has the lower latency in all cases. Notably, the latency from the simulation result is slightly higher than the real deployment. The reasons might be that the real execution time might not be the same as the profiled time, and there might be some caching from PyTorch. We also run the simulation on Resnet18 and Bert, as shown in Figure 4. The key takeaway is that for a larger model, the fast recovery mechanism can have a larger benefit since the wasted time is much smaller.

System	1 (# Failures)	2	5
FTPipeHD+EC	26.2s	25.8s	27.6s
PipeChurn	16.8s	20.2s	22.4s

Table 1: End-to-end latency with simultaneous failures

We also examine the performance of *PipeChurn* when we have simultaneous failures, as shown in Table 1, where *PipeChurn* can even have a larger improvement. Initially, we run the pipeline with 8 workers and later fail 1, 2, and 5 nodes simultaneously at time 10 to see the influence on the end-to-end training latency, respectively. When 5 nodes fail simultaneously, about 5/8 activations/gradients are lost, and it takes some time for the rest of the workers to recompute all the missing layers, and thus, the overall latency is larger than that of 1 and 2 failures.

5.2 Recovery delay

We then evaluate the time *PipeChurn* takes to resume the training process when churns happen. We focus on cases where multiple failures occur in a minibatch. This experiment is also done through training **Alexnet** of 1 epoch with the simulator. Table 2 shows the recovery delays of two approaches. Due to the fast recovery mechanism, PipeChurn has an average recovery delay of 3.1x smaller than FTPipeHD+EC.

System	1 (# Failures)	2	5
FTPipeHD+EC	11.5s	11.1s	12.3s
PipeChurn	2.7s	3.2s	7.2s

Table 2: Recovery delay comparison with FTPipeHD+EC

Similarly, we can notice that when there are 5 simultaneous failures, the recovery delay of PipeChurn increases largely. The reason lies in the partition result and the failed workers distribution (e.g., 1-5 workers); a particular worker may have to recompute most of the lost layers. More comprehensive results will be added in the final report.

5.3 Synchronization overhead

This section examines the synchronization overhead encountered in *PipeChurn*. We compare our solution with FTPipeHD, where each worker sends its weights to one of its neighbors and the central node for synchronization.

Table 3 shows the time taken by *PipeChurn* to update the weights at the end of each minibatch across 4 memory nodes. While the overall overhead diminishes with an increase in the number of workers, *PipeChurn* exhibits a more significant reduction ratio of approximately 1.6 from 3 workers to 10 workers, compared to FTPipeHD's reduction ratio of 1.2x.

Model	3 (# Workers)	5	10
Alexnet	2.22s	1.73s	1.36s
	4.97s	4.47s	4.1s
Resnet101	1.65s	1.28s	1.01s
	3.65s	3.28s	3s
BERT	19.17s	14.8s	11.52s
	43.69s	39.32	36.04

Table 3: Synchronization overhead comparison with FTPipeHD. The upper row of each cell is the result of *PipeChurn*

It should be noted that, theoretically, an increase in the number of memory nodes would reduce and more evenly distribute network bandwidth usage across each node. Consequently, this can lead to a further reduction in synchronization overhead. The final report will present a more comprehensive set of results.

6 Challenges

In this section, we describe some of the technical challenges of implementing PipeChurn.

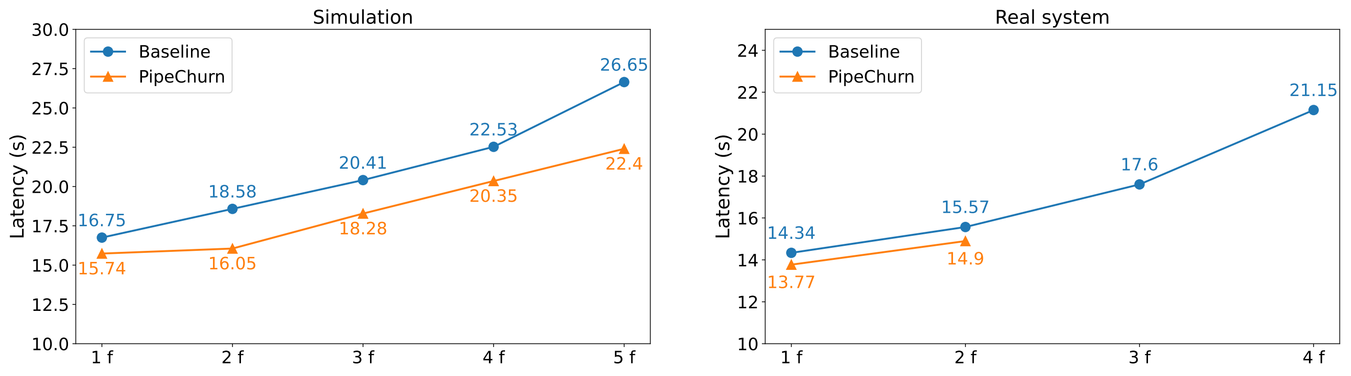


Figure 3: End-to-end latency comparison of training Alexnet for 1 Epoch.

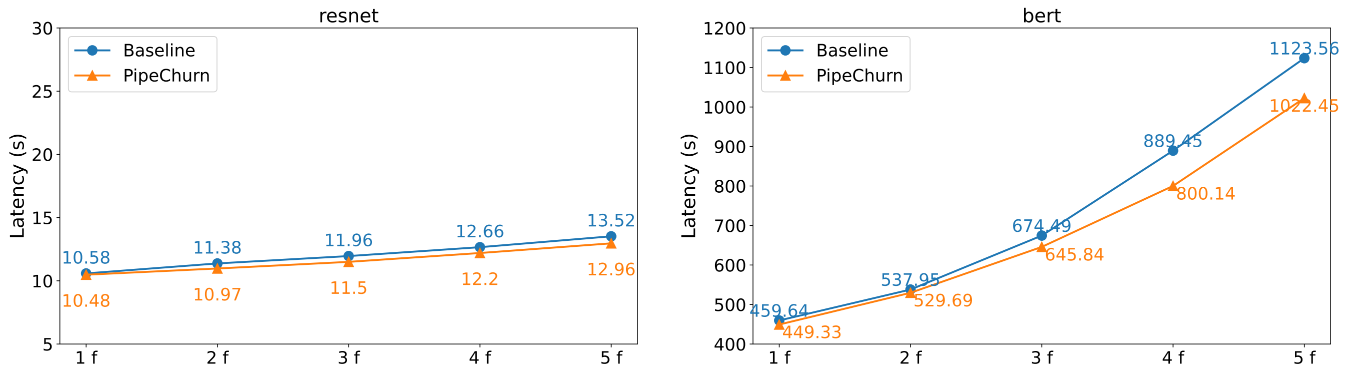


Figure 4: End-to-end latency comparison of training Resnet and Bert for 1 Epoch. The results are based on simulation.

6.1 PyTorch autograd limitations

Implementing PipeChurn with PyTorch poses significant challenges. Our goal is to manage the locations of activations and gradients distinctly. Activations produced by the model can be transferred to different devices due to churn, complicating the process. PyTorch's reliance on the **autograd** module complicates matters further, as it automates the creation of gradient computation details necessary for backward propagation following the forward execution. The core issue here is that these automatically generated details are not serializable, which hinders the possibility of performing the backward pass on activations that have been relocated.

6.2 PyTorch Backward Pass Profiling

Although PyTorch provides a *register_hook*() interface that allows users to access the backward's pre-hook in the torch.autograd.Variable class, this built-in function struggles with in-place activation functions like ReLU. Ideally, modifications to the PyTorch source code are necessary to achieve a more accurate measurement of backward pass running time. Due to these limitations, the profiling of model computation time may yield inaccurate results during the backward pass propagation. This inaccuracy could subsequently lead to errors in the following model partitioning process.

6.3 Inter-process communication

Each worker node operates with a dual-process system: a Golang process handles the membership list and communication with memory nodes, while a Python process manages the pipeline operations. Every time the churn happens or the workers do the minibatch synchronization, messages need to be relayed from the Python process on one worker node to the Golang process, and from there to the Golang process on the memory node. We use ZeroMQ to facilitate this inter-process communication and can potentially send complex data structures by using protobuf. Despite this, ensuring that both processes maintain a synchronized understanding of the pipeline's state remains a formidable challenge.

6.4 Memory measurement

During the profiling phase, we assess the memory requirements of each layer to ensure sufficient memory availability for each worker assigned to a specific stage. Yet, predicting the overall memory needs of a stage is challenging. For instance, the memory required by Alexnet's layers 1 through 19 totals approximately 6.5kB. However, profiling these layers collectively as a single stage shows a need for about 8.5kB of memory. This discrepancy can lead to the creation of an impractical partitioning plan for the system.

7 Related Work

Pipeline Parallelism. Deep learning models contain many layers, and pipeline parallelism naturally spreads these layers across multiple GPUs. GPipe [10] was among the first systems to propose this, splitting large models into several GPUs and employing synchronous pipeline training, where stochastic gradient descent (SGD) is performed until a mini-batch is complete. However, this approach incurs overhead and doesn't achieve full GPU utilization due to pipeline stages often being stalled, known as "bubble overhead."

To address this, PipeDream [17] and PipeMare [32] propose asynchronous pipelining for faster, more efficient training and improved utilization. Pipeline parallelism reduces the latency of gradient transmission between layers. PipeDream maintains statistical efficiency, reduces idle time, and overlaps communication with computation. It introduces "in pipeline" minibatches to prevent idle workers during DNN training. Key ideas include partitioning DNN layers into stages that progress at similar rates, ensuring enough minibatches to keep the pipeline full, and utilizing weight stashing for consistent weight versioning. While PipeDream is often referenced for partitioning, its optimality depends on accurate memory measurements; PipeMare avoids storing all forward weights, instead approximating them by retaining only weight velocity to reduce memory usage.

Fault-tolerant. FTPipeHD [6] is an asynchronous model similar to PipeDream but extended to heterogeneous edge devices, addressing challenges that IoT devices have different computing and memory capabilities, are connected via diverse wireless links, and frequently fail. In FTPipeHD, IoT devices hold the raw training data and a central node administrates. To provide fault tolerance, FTPipeHD combines chain replication with global replication in their weight distribution algorithm. Furthermore, FTPipeHD dynamically updates the optimal model partition points by estimating the time-varying computing resource of each heterogeneous worker periodically. While FTPipeHD is highly fault-tolerant, it suffers significant communication overhead in its weight redistribution.

Erasure Coding (EC). In large-scale deep-learning training, checkpoints are the primary approach for fault tolerance [16]. However, checkpointing requires the whole training process to be paused frequently and has to restart from the previous checkpoint when a failure happens. Erasure Coding, on the other hand, creates the in-memory redundancy to provide fault tolerance and save a significant space compared to replication. In [14], they use EC to encode the embedding tables and optimizer state. However, they only focus on the data parallelism model and the EC parameters where $r = 1$. That is, tolerating only 1 failure at a time.

Reconfiguration. In pipeline parallelism training, Varuna [4] utilizes dynamic configuration to optimize performance based on Spot Instance (SI). This involves scale-invariant calibration, comprising initial parameter determination via profiling and parameterized simulation to identify optimal configurations.

8 Future Work

In this section, we discuss some areas of future work to improve PipeChurn.

Churn rate. We believe it may be helpful to intelligently assign worker nodes based on their historical churn rates. Before adding a worker to a pipeline, performing profiling on its past history in *PipeChurn* can offer insights on its expected performance. Workers which fail more frequently might be less preferred for longer pipelines, while still offering sizable advantages for shorter pipelines.

Applying data parallelism. When we ran the experiments on Bert, we observed that a certain layer can have an execution time much longer than the others. This makes it impossible for the partitioner to find a balanced partition for the workers. One way to tackle this problem is by applying data parallelism to reduce the execution time for such layers. We expect in the future *PipeChurn* can take it into the consideration and optimize for it.

9 Conclusion

This paper presented the design and implementation of *PipeChurn*, a reliable pipeline parallelism framework. With the emergence of edge devices along with their computing power, *PipeChurn* enables users to execute a long-running model training job both adaptively and reliably. At its core, *PipeChurn* proposed a memory node architecture with erasure coding to provide fault tolerance and a fast recovery mechanism to minimize the impact of churns. We believe *PipeChurn* is a promising solution to accelerate AI development in edge environments.

References

[1] Gartner. https://www.gartner.com/en.

[2] Memberlist. https://github.com/hashicorp/memberlist.

[3] Athlur, S., Saran, N., Sivathanu, M., Ramjee, R., and Kwatra, N. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems* (New York, NY, USA, 2022), EuroSys '22, Association for Computing Machinery, p. 472–487.

[4] Athlur, S., Saran, N., Sivathanu, M., Ramjee, R., and Kwatra, N. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), pp. 472–487.

[5] Chang, Y., Wang, X., Wang, J., Wu, Y., Yang, L., Zhu, K., Chen, H., Yi, X., Wang, C., Wang, Y., et al. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology* (2023).

[6] Chen, Y., Yang, Q., He, S., Shi, Z., and Chen, J. Ftpipehd: A fault-tolerant pipeline-parallel distributed training framework for heterogeneous edge devices, 2021.

[7] Fan, S., Rong, Y., Meng, C., Cao, Z., Wang, S., Zheng, Z., Wu, C., Long, G., Yang, J., Xia, L., Diao, L., Liu, X., and Lin, W. Dapple: a pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2021), PPoPP '21, Association for Computing Machinery, p. 431–445.

[8] Hu, Y., Imes, C., Zhao, X., Kundu, S., Beerel, P. A., Crago, S. P., and Walters, J. P. N. Pipeline parallelism for inference on heterogeneous edge computing, 2021.

[9] Hua, H., Li, Y., Wang, T., Dong, N., Li, W., and Cao, J. Edge computing with artificial intelligence: A machine learning perspective. *ACM Comput. Surv. 55*, 9 (jan 2023).

[10] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[11] Lamy-Poirier, J. Breadth-first pipeline parallelism, 2022.

[12] Li, P., Koyuncu, E., and Seferoglu, H. Respipe: Resilient model-distributed dnn training at edge networks. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2021), pp. 3660–3664.

[13] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S. Pytorch distributed: Experiences on accelerating data parallel training, 2020.

[14] Liu, K., Kosaian, J., and Rashmi, K. V. ECRM: Efficient Fault Tolerance for Recommendation Model Training via Erasure Coding. *arXiv* (Apr. 2021).

[15] López Escobar, J. J., Díaz Redondo, R. P., and Gil-Castiñeira, F. In-depth analysis and open challenges of mist computing. *Journal of Cloud Computing 11*, 1 (Nov. 2022).

[16] Maeng, K., Bharuka, S., Gao, I., Jeffrey, M., Saraph, V., Su, B.-Y., Trippel, C., Yang, J., Rabbat, M., Lucia, B., et al. Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. *Proceedings of Machine Learning and Systems 3* (2021), 637–651.

[17] Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 1–15.

[18] Nezami, Z., Zamanifar, K., Djemame, K., and Pournaras, E. Decentralized edge-to-cloud load balancing: Service placement for the internet of things. *IEEE Access 9* (2021), 64983–65000.

[19] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (USA, 2014), USENIX ATC'14, USENIX Association, p. 305–320.

[20] Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models, 2020.

[21] Rojas, E., Kahira, A. N., Meneses, E., Gomez, L. B., and Badia, R. M. A study of checkpointing in large scale training of deep neural networks, 2021.

[22] Shallue, C. J., Lee, J., Antognini, J., Sohl-Dickstein, J., Frostig, R., and Dahl, G. E. Measuring the effects of data parallelism on neural network training. *Journal of Machine Learning Research 20*, 112 (2019), 1–49.

[23] Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[24] Su, X., Wei, B., Wu, Y., and Wu, J. Efficient erasure-coded data update and recovery based on machine learning and i/o mitigation. In *2022 IEEE 24th Int Conf on High Performance Computing Communications; 8th Int Conf on Data Science Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud Big Data Systems Application (HPCC/DSS/SmartCity/DependSys)* (2022), pp. 1141–1146.

[25] Thorpe, J., Zhao, P., Eyolfson, J., Qiao, Y., Jia, Z., Zhang, M., Netravali, R., and Xu, G. H. Bamboo: Making preemptible instances resilient for affordable training of large dnns, 2022.

[26] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[27] Wang, Z., Jia, Z., Zheng, S., Zhang, Z., Fu, X., Ng, T. S. E., and Wang, Y. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 364–381.

[28] Wang, Z., Jia, Z., Zheng, S., Zhang, Z., Fu, X., Ng, T. S. E., and Wang, Y. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Oct. 2023), SOSP '23, ACM.

[29] Wei, B., Wu, J., Su, X., Huang, Q., Liu, Y., and Zhang, F. Efficient erasure-coded data updates based on file class predictions and hybrid writes. *Computers and Electrical Engineering 104* (2022), 108441.

[30] Wu, J., Gan, W., Chen, Z., Wan, S., and Philip, S. Y. Multimodal large language models: A survey. In *2023 IEEE International Conference on Big Data (BigData)* (2023), IEEE, pp. 2247–2256.

[31] Xu, A., Huo, Z., and Huang, H. On the acceleration of deep learning model parallelism with staleness. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020), pp. 2085–2094.

[32] Yang, B., Zhang, J., Li, J., Ré, C., Aberger, C., and De Sa, C. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems 3* (2021), 269–296.

[33] Yi, S., Hao, Z., Qin, Z., and Li, Q. Fog computing: Platform and applications. In *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)* (Nov. 2015), IEEE.

[34] Yin, Y., and Deng, L. A dynamic decentralized strategy of replica placement on edge computing. *International Journal of Distributed Sensor Networks 18*, 8 (Aug. 2022), 155013292211150.

[35] Yoon, J., Byeon, Y., Kim, J., and Lee, H. Edgepipe: Tailoring pipeline parallelism with deep neural networks for volatile wireless edge devices. *IEEE Internet of Things Journal 9*, 14 (2022), 11633–11647.

[36] Zhao, W. X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., and Dong, Z. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).