# Gemini: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints

Zhuang Wang*
Rice University

Zhen Jia
Amazon Web Services

Shuai Zheng
Amazon Web Services

Zhen Zhang
Amazon Web Services

Xinwei Fu
Amazon Web Services

T. S. Eugene Ng
Rice University

Yida Wang
Amazon Web Services

## Abstract

Large deep learning models have recently garnered substantial attention from both academia and industry. Nonetheless, frequent failures are observed during large model training due to large-scale resources involved and extended training time. Existing solutions have significant failure recovery costs due to the severe restriction imposed by the bandwidth of remote storage in which they store checkpoints.

This paper presents Gemini, a distributed training system that enables fast failure recovery for large model training by checkpointing to CPU memory of the host machines with much larger aggregated bandwidth. However, two challenges prevent naïvely checkpointing to CPU memory. First, the availability of checkpoints in CPU memory cannot be guaranteed when failures occur. Second, since the communication traffic for training and checkpointing share the same network, checkpoint traffic can interfere with training traffic and harm training throughput. To address these two challenges, this paper proposes: 1) *a provably near-optimal checkpoint placement strategy* to maximize the probability of failure recovery from checkpoints in CPU memory; and 2) *a checkpoint traffic scheduling algorithm* to minimize, if not eliminate, the interference of checkpoint traffic on model training. Our evaluation shows that overall Gemini achieves a faster failure recovery by more than 13× than existing solutions. Moreover, it achieves optimal checkpoint frequency, i.e., every iteration, and incurs no overhead on training throughput for large model training.

*Work done during Zhuang's internship at Amazon Web Services.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*SOSP '23, October 23–26, 2023, Koblenz, Germany*
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0229-7/23/10…$15.00
https://doi.org/10.1145/3600006.3613145

*CCS Concepts:* • **Computer systems organization → Dependable and fault-tolerant systems and networks**; • **Computing methodologies → Machine learning**.

***Keywords:*** Distributed Training, Fault Tolerance, In-memory Checkpoint

**ACM Reference Format:**
Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. Gemini: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3600006.3613145

## 1 Introduction

Deep learning models have shown their ability to perform outstandingly on a spectrum of tasks including computer vision [33, 69], natural language processing [26, 76], etc. Recently, language models like ChatGPT [8] and GPT-4 [53] have drawn significant attention from both academia and industry with unprecedented performance as well as model size. PaLM [24] has 540 billion parameters, which is a 360× increase over GPT-2 [61] that was released three years earlier. This trend is still expediting because continued improvements have been observed from scaling the model sizes [24]. To train such a large model, failures are inevitably frequent because of the number of involved accelerators (e.g., tens of thousands of GPUs) and the length of training time (in months). For example, OPT model training reports a failure frequency of twice a day [14]. The situation will get worse as the model size keeps growing.

Existing solutions cannot handle training failures efficiently. According to the report from OPT-175B training [85], about 178,000 GPU hours were wasted due to various training failures. As the failure frequency increases with the scale of the training, failures can dramatically slow down the training progress (up to 43% [44]). One major reason for such a significant overhead caused by failures is the inefficiency of checkpointing. Existing solutions rely on naïve checkpointing [3, 28, 48], which periodically saves the model states to a remote persistent storage system, for *failure recovery*, i.e., the process to fetch the latest checkpoint and resume training to the states right before a failure. Intuitively, a higher network bandwidth leads to shorter checkpoint retrieval times, and a higher checkpoint frequency reduces training progress loss in case of failures. However, existing solutions are restricted by the low bandwidth to remote persistent storage, resulting in significant failure recovery costs, i.e., taking up to tens of minutes to retrieve the checkpoint captured a few hours ago to resume the training. It is worth noting that the state-of-the-art large model training adopts a synchronized method to guarantee model quality [50, 83, 85], making it infeasible to only drop the training progress of the failed machine/device upon a failure to proceed without waiting for the failure recovery. Instead, it requires all machines/devices to roll back to the same checkpoint for failure recovery.

To reduce the prohibitively large failure recovery overhead, this paper presents Gemini, a distributed training system that leverages the high bandwidth of CPU memory to achieve fast failure recovery in large model training via prompt checkpoint retrieval (in seconds) and high checkpoint frequency (ideally checkpoint for every training iteration). Gemini incorporates the hierarchical storage consisting of local CPU memory, remote CPU memory, and remote persistent storage, to store checkpoints. It leverages CPU memory to store checkpoints for failure recovery, and meanwhile stores checkpoints for other purposes in remote persistent storage. Gemini takes advantage of the optimized network connection for large-scale training to checkpoint the model states in the CPU memory of the compute cluster, which allows for a much higher frequency than existing solutions. It guarantees a 100% failure recovery and always fetches the available checkpoint from the fastest storage to minimize the recovery cost.

Checkpointing to CPU memory raises two questions that Gemini needs to address. First, **how to maximize the probability of a successful failure recovery from CPU memory?** The availability of checkpoints in CPU memory is not guaranteed upon a failure as the corresponding machines could be down. When the checkpoints are unavailable, in the worst case, the system has to resort to checkpoints stored in remote persistent storage, leading to significant failure recovery costs. The success rate of recovering a failure from checkpoints stored in CPU memory largely depends on how the checkpoints are placed among the CPU memory in different host machines. Gemini stores redundant checkpoints and proposes a placement strategy that maximizes the probability. We have proved that the strategy is optimal when the number of machines participating in training is divisible by the number of replicas and the strategy remains near-optimal with established bounds in other cases. Second, **how to minimize the interference of checkpoint traffic with model training?** Checkpointing model states to remote CPU memory shares the network resource with the regular training. Naïvely checkpointing to CPU memory will easily delay the training traffic which impacts the training throughput. Gemini designs a deliberate communication scheduling algorithm for interleaving these two types of traffic to minimize the interference on training throughput.

Gemini makes no assumptions about the underlying parallelism strategy [42, 49, 62, 67, 89] of the training system. It targets static and synchronous training with fixed computation resources, following the common practice for large model training in industrial settings [3, 24, 28, 68, 77, 78]. Elastic training [45, 54, 81] and asynchronous training [47, 84] are beyond the scope of this paper. Gemini also makes no assumptions about the accelerator. In this paper, we conducted experiments on NVIDIA GPUs, but the technique applies to other accelerators such as AWS Trainium [2], which remains for future work.

To sum up, this paper makes the following contributions:

- To the best of our knowledge, Gemini is the first system that takes advantage of CPU memory checkpointing to achieve efficient failure recovery in large model training.
- We design a provably near-optimal checkpoint placement strategy that maximizes the probability of a successful failure recovery from CPU memory.
- We propose a communication scheduling algorithm that pipelines checkpoint traffic across host machines to minimize its interference with model training.

We build Gemini atop DeepSpeed [63] and evaluate it with ZeRO-3 [62] on various large deep learning models using both Amazon EC2 p4d.24xlarge (NVIDIA A100 GPUs) and p3dn.24xlarge (NVIDIA V100 GPUs) instances. Compared to existing solutions [3, 48], Gemini reduces the checkpoint retrieval time by up to 250× and improves the checkpoint frequency by up to 8×. As a result, Gemini achieves a faster failure recovery by more than 13× without incurring overhead on training throughput.

## 2 Motivation

### 2.1 Failure Recovery in Model Training

**Frequent failures in model training.** Developers have observed many failures during large model training due to the large number of GPUs and the long training time. For example, training OPT-175B used 992 NVIDIA A100 GPUs, and the training process encountered around 110 failures over a period of two months [85]. Similar symptoms have also been reported during training BLOOM [3].

**Wasted time for failure recovery.** We have noticed a significant waste of computation resources caused by large model training failures. The *model states*, i.e., the learnable parameters and the optimizer states, are resided in GPU memory during training. When a failure occurs, the model states must be rolled back to previous states by retrieving

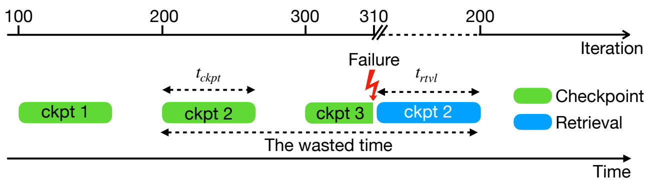


**Figure 1.** An illustration of how failure recovery uses checkpoints. The checkpoint frequency $f$ to the remote persistent storage is every 100 iterations (same as BLOOM [3]). A failure occurs at iteration 310 when the third checkpoint is incomplete. The failure recovery rolls back the model states to iteration 200 by retrieving the second checkpoint.

the latest checkpoint for failure recovery. For example in Figure 1, a failure occurs at iteration 310, but the latest available checkpoint is at iteration 200. After the failure recovery, the training progress from iteration 200 to 310 is lost. Additionally, retrieving the latest checkpoint incurs overhead during the failure recovery process.

We define *wasted time* as the sum of the time spent on the lost training process before a failure and the time for retrieving the latest checkpoint during a failure recovery. As illustrated in Figure 1, the wasted time describes the timespan of a paused training process due to a failure, i.e., the time of computation resource wasted in terms of the training process. It is determined by three factors:

- *checkpoint time*, which is the time to finish a checkpoint of model states. We denote checkpoint time as $t_{ckpt}$ in Figure 1.
- *checkpoint frequency*, which determines how frequently the training system checkpoints the model states to the storage system. We denote checkpoint frequency as $f$.
- *retrieval time*, which is the time to retrieve the latest complete checkpoint[1]. We denote retrieval time as $t_{rtvl}$, shown in Figure 1.

In this paper, we use the average wasted time as the main metric to evaluate the performance of a checkpointing solution, because a failure may occur at any time and the wasted time varies. The best case is that a failure occurs right after the completion of a checkpoint and the wasted time is $t_{cpkt} + t_{rtvl}$. The worst case is that a failure occurs right before the completion of a checkpoint and the wasted time is $t_{cpkt} + 1/f + t_{rtvl}$. Assuming failures are evenly distributed between two consecutive checkpoints, the average wasted time (denoted as $T_{wasted}$) can be expressed as

$$T_{wasted} = t_{ckpt} + \frac{1}{2f} + t_{rtvl}. \tag{1}$$

In addition, we have the following constraint:

$$1/f \geq max(t_{ckpt}, T_{iter}), \tag{2}$$

[1]We exclude the overheads to fix failures and replace machines in the wasted time because they are not caused by checkpoints.

| Instance type | Cloud | GPU | GPU memory | CPU memory |
|---|---|---|---|---|
| p3dn.24xlarge [15] | AWS | 8 V100 | 8 × 32 GB | 768 GB |
| p4d.24xlarge [16] | AWS | 8 A100 | 8 × 40 GB | 1152 GB |
| ND40rs_v2 [11] | Azure | 8 V100 | 8 × 32 GB | 672 GB |
| ND96asr_v4 [12] | Azure | 8 A100 | 8 × 40 GB | 900 GB |
| n1-8-v100 [10] | GCP | 8 V100 | 8 × 32 GB | 624 GB |
| a2-highgpu-8g [10] | GCP | 8 A100 | 8 × 40 GB | 640 GB |
| DGX A100 [13] | NVIDIA | 8 A100 | 8 × 80 GB | 2 TB |

**Table 1.** The CPU memory size is much larger than the GPU memory size in GPU machines.

where $T_{iter}$ is the iteration time. One checkpoint cannot start until its previous checkpoint completes, and there is no need to have multiple checkpoints within one iteration as the model states are updated once every iteration.

To reduce the wasted time, it is critical to reduce checkpoint time $t_{ckpt}$ to enable a higher checkpoint frequency $f$, and the optimal frequency $f$ is every iteration $1/T_{iter}$.

## 2.2 Limitations of Existing Solutions

Existing solutions fail to achieve high checkpoint frequency for failure recovery due to the remote persistent storage system usage. They checkpoint the model states at a particular frequency and persist checkpoints in a remote persistent storage system [48, 65]. In common practice, existing solutions checkpoint model states at a low frequency, e.g., every three hours in BLOOM training [3], to reduce the required storage capacity. A few hours of computation resources are wasted when a failure occurs. Considering thousands of GPUs involved in training and hundreds of failures experienced during training, the total computation resource waste is significant, and the training time slowdown can be up to 43% [44]. It is infeasible to arbitrarily increase the checkpoint frequency because checkpoint frequency is bottlenecked by the bandwidth of the remote persistent storage [28]. For example, it takes 42 minutes to checkpoint the model states of MT-NLG [68] to the remote persistent storage when the bandwidth is 20Gbps. According to Equation (1), the average wasted time for failure recovery is 105 minutes, which makes the training system less efficient.

## 2.3 The Opportunity and Challenges

Minimizing the wasted time for failure recovery is crucial for enhancing the system efficiency of distributed training, especially large model training. We next explore the opportunity to achieve this goal and discuss the identified challenges.

**2.3.1 Checkpointing to CPU memory.** The low bandwidth severely restricts the frequency of checkpointing to remote persistent storage. We observe that the CPU memory in GPU machines is sufficient to store a few checkpoints. Table 1 compares the GPU and CPU memory in popular GPU instances in public clouds for large model training, demonstrating that the CPU memory is much larger than the

GPU memory. This observation provides a great opportunity for Gemini to store the latest checkpoint in CPU memory. Gemini can leverage the network connecting GPU instances for checkpointing. Because this network is optimized for training, its bandwidth is much higher than the bandwidth of the remote persistent storage [16]. Therefore, Gemini can achieve a much higher checkpoint frequency for failure recovery than existing solutions.

One concern is that the CPU memory size is insufficient to store the history of checkpoints for purposes other than failure recovery, such as transfer learning [56] and model debugging [23, 28]. To address this concern, Gemini decouples checkpoints for different purposes. It only stores the checkpoints for failure recovery in CPU memory, while storing checkpoints for other purposes in remote persistent storage.

**2.3.2 Challenges.** Checkpointing to CPU memory allows for a much higher frequency than existing solutions, thereby reducing the wasted time. However, this approach also presents new challenges.

**How to maximize the probability of failure recovery from checkpoints stored in CPU memory?** Although checkpointing to CPU memory enables a high frequency, the availability of checkpoints in CPU memory cannot be guaranteed when failures occur. In the cases of unavailable checkpoints in CPU memory, we have to fall back to using the low-frequency checkpoints stored in the remote persistent storage for failure recovery, causing significant wasted time.

**How to minimize the interference of checkpoint traffic with model training?** When storing checkpoints in CPU memory, communication traffic for training and checkpointing have to share the same network. Without careful design, checkpoint traffic can interfere with training traffic and harm training throughput. The interference overhead is non-negligible because it can negatively impact every iteration. This can significantly diminish the benefits gained from checkpointing to CPU memory.

## 3 System Architecture of Gemini

We propose Gemini, which achieves a high checkpoint frequency, even every iteration, to optimize the failure recovery overhead in distributed training. It minimizes the wasted time by checkpointing to CPU memory and addresses the two aforementioned challenges. Figure 2 illustrates Gemini's architecture that consists of two modules: 1) a checkpoint creation module (Section 3.1); and 2) a failure recovery module (Section 3.2). The two modules cooperate to resume training once a failure occurs.

### 3.1 Checkpoint Creation Module

Gemini uses a decoupled and hierarchical storage design for checkpointing. In Gemini, the checkpoint creation module stores the checkpoints of each GPU machine to different destinations, including local CPU memory, remote CPU memory

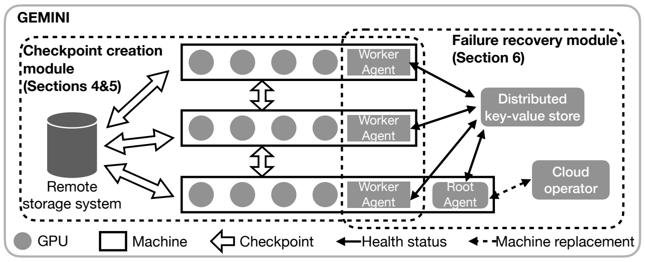


**Figure 2.** The system architecture of Gemini. Gemini consists of checkpoint creation and failure recovery modules. In the checkpoint creation module, each worker agent controls checkpoint destinations and schedules checkpoint communications. In the failure recovery module, worker agents update machines' health statuses in the distributed key-value store. The root agent periodically checks the health statuses in the distributed key-value store, interacts with the cloud operator to replace failed machines as needed, and guides the checkpoint retrieval for failure recovery.

on other machines, and remote persistent storage. The checkpoint creation module stores the checkpoints for failure recovery in local and remote CPU memory. These checkpoints are managed by Gemini's checkpoint creation module and are transparent to users. On the other hand, checkpoints for other purposes, such as transfer learning [56] and model debugging [28], are stored in remote persistent storage and managed by users. During failure recovery, checkpoints are first retrieved from local CPU memory and then remote CPU memory if unavailable in local CPU memory. If both local and remote CPU memory checkpoints are unavailable, Gemini retrieves checkpoints from remote persistent storage.

As illustrated in Figure 2, each training machine has a Gemini worker agent for checkpointing to CPU memory. Where to place checkpoints for failure recovery on CPU memory determines the failure recovery capacity. To maximize the probability of failure recovery from checkpoints in CPU memory, we propose a provably near-optimal checkpoint placement strategy for checkpointing to CPU memory (Section 4). Gemini determines the checkpoint placement strategy when training is initialized. During runtime, the Gemini worker agent on each machine communicates checkpoints from GPU memory to CPU memory based on the placement strategy and checkpoint frequency. To minimize or even eliminate the interference of checkpoint traffic with model training, we propose a traffic scheduling algorithm that pipelines checkpoint traffic and interleaves it with training traffic (Section 5).

### 3.2 Failure Recovery Module

Gemini's failure recovery module has four components: a group of Gemini worker agents, a Gemini root agent, a distributed key-value store, and a cloud operator. Worker agents monitor their own machine's health status and update it in the distributed key-value store [9, 29, 30]. The unique root agent runs on a regular training machine with a worker agent. The training machine with the root agent running is called the root machine. The root agent periodically checks the health status of each training machine from the distributed key-value store. The cloud operator manages the training computation resources and replaces failed machines with healthy ones as needed.

If the root agent detects a training machine failure, the root agent takes corresponding actions based on failure types (Section 6). For example, when a training machine replacement is needed, the root agent interacts with the cloud operator to complete the machine replacement and guides the replaced machine where to retrieve its checkpoints.

Worker agents also periodically check the root machine's health status in the distributed key-value store. A root machine failure is detected when the root machine's health status has not been updated for a predefined time threshold. In the case of a root machine failure, one alive worker machine is promoted as the root machine, and one new worker machine is initialized to replace the failed one. Gemini relies on the leader election method in the distributed key-value store [40, 52] for the root machine selection.

## 4 Checkpoint Placement to CPU Memory

To reduce the wasted time, Gemini writes checkpoints to CPU memory to achieve high frequencies. However, the checkpoints stored in CPU memory may become invalid for recovery when some GPU machines are disconnected from training. In such cases, Gemini has to fetch from remote persistent storage to perform recovery, leading to significant wasted time. Adding more checkpoint replicas reduces the possibility of unavailable checkpoints in CPU memory, but it also increases CPU memory usage and network bandwidth competition with training traffic. In addition to the number of replicas, our research has revealed that the checkpoint placement strategy, i.e., where to store the checkpoint replicas, also affects the possibility, as shown in Figure 3. Hence, we aim to identify the best placement strategy that maximizes the probability of failure recovery from CPU memory, given a specific number of replicas. This problem can be formulated as follows:

**Problem 1.** *Given N machines and m checkpoint replicas, what is the optimal placement strategy to distribute the m replicas among the N machines to maximize the probability of failure recovery from CPU memory?*

We design a mixed placement strategy described in Algorithm 1 to solve Problem 1. The inputs of the algorithm are the number of machines $N$, and the number of replicas $m$. The output is the machine group assignment and the specific strategy. If the number of machines $N$ is divisible

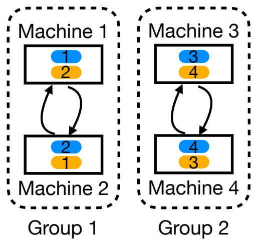
**Algorithm 1:** Mixed checkpoint placement strategy

```
Input: N is the number of GPU machines and m is the number of
       checkpoint replicas.
Output: The group list $\mathcal{G}$ and the strategy.
Function placement_strategy(N, m):
    $\mathcal{G}$ = []
    $g = \lfloor N/m \rfloor$
    for $i \leftarrow 0$ to $g - 1$ do
        $G$ = []
        for $j \leftarrow 1$ to $m$ do
            $G.add(m \times i + j)$
        end
        $\mathcal{G}.add(G)$
    end
    $strategy$ = "group"
    if N is not divisible by m then
        $strategy$ = "mixed"
        // add remaining machines to the last group
        for $j \leftarrow g \times m + 1$ to $N$ do
            $\mathcal{G}[-1].add(j)$
        end
    end
    return $\mathcal{G}$, $strategy$
```

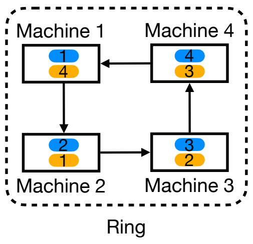
by the number of replicas $m$, we will apply a group placement strategy for all machines participating in training. The $N$ machines are divided into $N/m$ groups and each group has $m$ machines. During training, each machine broadcasts its checkpoints to the $m-1$ machines in the same group. It also writes one checkpoint to its own CPU memory as a local replica, which is one tier in Gemini's hierarchical checkpoint solution. Otherwise, when $N$ is not divisible by $m$, we split the $N$ machines into $\lfloor N/m \rfloor$ groups and apply the group placement strategy to the first $\lfloor N/m \rfloor - 1$ groups. For the last $N - m(\lfloor N/m \rfloor - 1)$ machines, we apply a ring placement strategy, in which each machine writes the checkpoints from GPU memory to its local CPU memory and also sends checkpoints to the consecutive $m-1$ machines in the ring from its left hand. Regardless of the placement strategy employed, Gemini copies the checkpoint from GPU memory to the local CPU memory and treats it as a local replica. It has two advantages: 1) it can mitigate the network bandwidth contention with training traffic; and 2) for certain failure types, e.g., software failures (refer to Section 6.1), Gemini can directly resume training from the local replica to accelerate checkpoint retrieval. We pivot the group placement strategy because it exhibits a greater likelihood of recovering from CPU memory compared to the ring placement strategy with the same number of replicas. We also have Theorem 1 for the performance of the mixed placement strategy. Refer to Appendix A for the proof.

**Theorem 1.** *To address Problem 1 for checkpoint placement:*

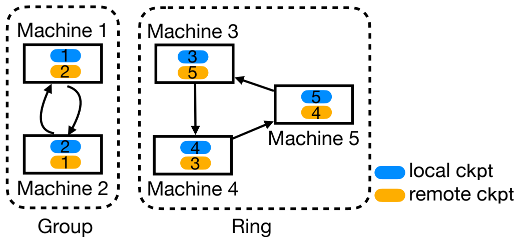
1. *When N is divisible by m, the mixed placement strategy (equals group placement strategy) is the optimal placement strategy.*



**(a)** Group placement strategy.



**(b)** Ring placement strategy.



**(c)** Mixed placement strategy.

**Figure 3.** Illustrations of the mixed checkpoint placement strategy.

2. *When $N$ is not divisible by $m$, the mixed placement strategy minimizes the checkpoint communication time. Its failure recovery probability from CPU memory is near-optimal and the gap is bounded by $(2m-3)/\binom{N}{m}$.*

Figure 3a illustrates an example of the group placement strategy with $N = 4$ and $m = 2$. There are two groups and each group has two machines. Each machine has a local checkpoint, i.e., its local machine checkpoint, and a remote checkpoint, i.e., the checkpoint from the other machine in the same group. Figure 3b illustrates an example of the ring placement strategy with $N = 4$ and $m = 2$, in which all machines form a ring structure for checkpointing to CPU memory. Assume two machines fail at the same time. With the group placement strategy, training can recover failures from CPU memory except Machines 1 and 2, or Machines 3 and 4 fail simultaneously (a total of two possible cases). However, with the ring placement strategy, the concurrent failures of any two consecutive machines (four possible cases in total) will result in the loss of both replicas of a checkpoint stored in CPU memory. Consequently, the probability that training has to fetch remote persistent storage for failure recovery with the group placement strategy is 50% lower than that with the ring placement strategy. Figure 3c also illustrates an example of the mixed placement strategy with $N = 5$ and $m = 2$, in which the first two machines form a group and the last three machines form a ring.

With the group placement strategy, we calculate the probability that Gemini can recover failures from CPU memory using Corollary 1. Refer to Appendix B for the proof. According to Corollary 1, when the number of machines $N$ is 16, the number of replicas $m$ is 2, and the failure machine $k$ is 2, the probability is 93.3% and it increases with $N$. It means that with two checkpoint replicas, Gemini can resume training from CPU memory in most cases.

**Corollary 1.** *When $N$ is divisible by $m$ and $k$ machines are disconnected simultaneously, the probability that Gemini can recover failures from CPU memory is*

$$\begin{cases} \Pr(N,m,k) = 1, & \text{if } k < m \\ \Pr(N,m,k) \geq \max\{0, 1 - \frac{N\binom{N-m}{k-m}}{m\binom{N}{k}}\}, & \text{if } m \leq k \leq N \end{cases} \tag{3}$$

## 5 Minimizing Training Interference

Frequently writing checkpoints to remote CPU memory might hinder overall training performance due to potential network bandwidth competition with training traffic. Our primary objective is to minimize the wasted time without compromising training performance. In this section, we will explain how Gemini mitigates the interference caused by frequent checkpointing. We begin by examining the possibility of minimizing the impact of checkpointing on model training (Section 5.1), then discussing the challenges and the approaches we took to overcome them (Section 5.2). Finally, we elaborate on the specific algorithm and mechanism we used in Gemini (Section 5.3 & 5.4).

### 5.1 Traffic Interleaving

Modern distributed training, such as large model training, relies on collective communication operations for synchronization. For example, in ZeRO [62], each GPU needs to fetch the parameters of each layer from other GPUs before its computation in both forward and backward passes. These communication operations can block computation when the parameters of a layer are not ready but the computation of the previous layer has been completed. We denote the communication traffic for model computation, including gradient synchronization and parameter fetching, as *training traffic*. An example of training traffic during model computation is shown in Figure 4a. When checkpointing to remote CPU memory, its traffic, denoted as *checkpoint traffic*, shares the same network as training traffic, resulting in potential network resource contention that may delay training traffic and hinder computations. When performing checkpointing at the start of subsequent iterations, it blocks the training process and incurs non-negligible overheads for model training, as illustrated in Figure 4b. This significantly negates the benefits gained from the reduced wasted time by checkpointing to CPU memory. Hence, Gemini must carefully orchestrate the checkpoint traffic to minimize its interference with training throughput. Fortunately, we observe that the network has idle timespans overlapped with computation and this naturally occurs in large model training. This observation

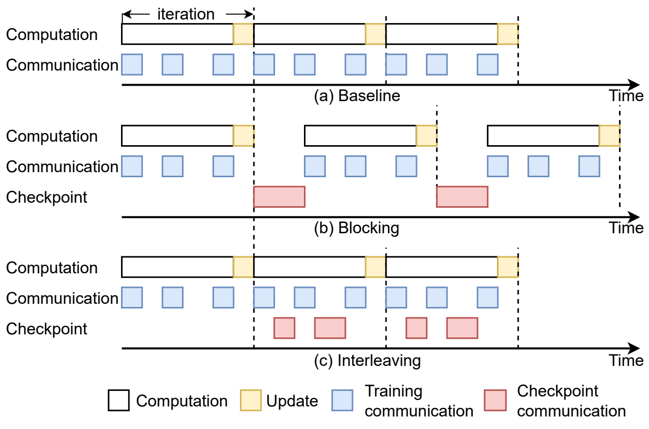


**Figure 4.** Interleaving communications of training and checkpointing can minimize the interference.

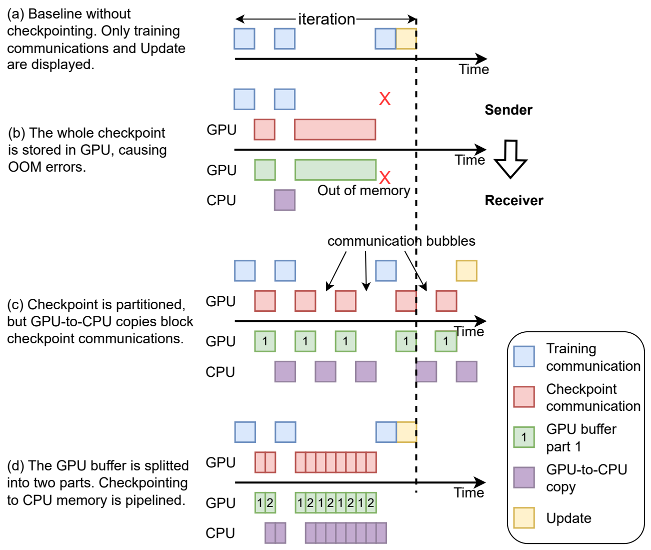

(a) Baseline without checkpointing. Only training communications and Update are displayed.
iteration
Time
(b) The whole checkpoint is stored in GPU, causing OOM errors.
GPU
GPU
CPU
Time
Out of memory
Sender
Receiver
communication bubbles
(c) Checkpoint is partitioned, but GPU-to-CPU copies block checkpoint communications.
GPU
GPU
CPU
Time
(d) The GPU buffer is splitted into two parts. Checkpointing to CPU memory is pipelined.
GPU
GPU
CPU
Time
Training communication
Checkpoint communication
GPU buffer part 1
GPU-to-CPU copy
Update

**Figure 5.** Different schemes for interleaving training and checkpoint traffic.

provides a great opportunity for Gemini to insert checkpoint traffic in these idle timespans and overlap checkpoint communications with computation, as shown in Figure 4c.

## 5.2 Difficulties and Approaches

Gemini needs to write checkpoints from local GPU memory to CPU memory on remote machines. It first uses GPU-to-GPU communications to send checkpoints between machines for interleaving checkpoint traffic with training traffic, which also uses direct GPU-to-GPU communications [41, 64] among machines in large model training [37, 60, 87]. After that, it transmits the checkpoints from GPU memory on remote machines to their CPU memory with GPU-to-CPU copy. This design allows scheduling training traffic and checkpoint traffic in the application layer without relying on the network layer. Gemini orchestrates both types of traffic by leveraging existing inter-GPU communication libraries, such as NCCL [1], in a unified manner. However, this design raises two practical difficulties.

**Difficulty: Extra GPU memory consumption.** Naïvely sending a whole checkpoint from a local GPU to a remote GPU consumes a significant amount of GPU memory, which may trigger GPU out-of-memory (OOM) and crash the training process, as shown in Figure 5b. The checkpoint size is huge in large model training. For example, the checkpoint size of GPT2-100B [87] on each GPU is 9.4GB. Furthermore, most GPU memory has already been used to store model parameters, gradients, and intermediate results. Therefore, a remote GPU is unlikely to accommodate a whole checkpoint during large model training.

**Approach: Partitioning checkpoint.** Although a whole checkpoint with several GBs is too large for a remote GPU, we observe that each GPU usually has a few hundred of memory available during training based on our profiling results. Gemini first reserves a small GPU memory buffer for checkpoint communications, then partitions a whole checkpoint into small chunks and transfers the small chunks separately. The remote GPU moves the received chunk to CPU memory once a communication completes making the buffer available for the next communication. Figure 5c illustrates the process of partitioning checkpoint.

**Difficulty: Local GPU-to-CPU copy overhead.** Checkpointing to remote CPU memory includes a procedure of GPU-to-CPU copy on the receiver side. The sender cannot transit new checkpoint chunks until the GPU-to-CPU copy is complete, causing communication bubbles in the GPU-to-GPU communication timeline, as shown in Figure 5c. Since the GPU-to-CPU memory copy bandwidth is comparable to the inter-machine GPU-to-GPU network bandwidth [2], the bubble time could be close to the inter-machine GPU-to-GPU checkpoint communication time, which may exacerbate the interference with model training.

**Approach: Pipelining checkpoint transmission.** Gemini uses a pipeline mechanism to allow checkpoint communications to fully leverage the network idle timespans. It splits the reserved GPU memory buffer into multiple sub-buffers and partitions the checkpoints into chunks that fit into these sub-buffers. Gemini alternatively uses these sub-buffers for transferring checkpoint chunks. When copying a chunk from GPU to CPU memory, Gemini can simultaneously receive a new checkpoint chunk using GPU-to-GPU communication in a separate sub-buffer. Figure 5d illustrates an example with two sub-buffers. Inter-machine GPU-to-GPU communication overlaps with local GPU-to-CPU memory copy and the idle timespans are fully utilized for checkpoint traffic.

[2]We measured both bandwidths in our p4d.24xlarge instances in AWS and both are around 400Gbps.

**Algorithm 2:** Checkpoint Partition Algorithm

```
Input: $\mathcal{T} = \{t_1, t_2, \ldots, t_d\}$ is the set of idle timespans. $C$ is the
       size of a checkpoint and $m - 1$ is the number of
       checkpoints for communications. There are $p$ buffer parts
       and the size of each part is $R/p$. $B$ is the network
       bandwidth. $\mu \in (0, 1)$ is a coefficient for the variance of idle
       spans across iterations. $f(s)$ is the communication time for
       a checkpoint chunk with size $s$
Output: The checkpoint partitions.
1  Function checkpoint_partition():
2      $t[d] = +\infty$
3      $partitions = [\,]$
4      $cpkt\_id = 0$
5      $remain\_size = C$
6      foreach $t \in \mathcal{T}$ do
7          $remain\_span = \mu \times t$
8          while $remain\_span > 0$ do
9              if $remain\_span \geq f(R/p)$ then
10                 $size = R/p$
11             else
12                 $size = \max\{0, (remain\_span - \alpha)B\}$
13             end
14             $size = \min\{remain\_size, size\}$
15             if $size > 0$ then
16                 $remain\_size = remain\_size - size$
17                 $remain\_span =$
                     $remain\_span - f(remain\_size)$
18                 $partitions$.add($size$)
19             end
20             if $remain\_size == 0$ then
21                 if $cpkt\_id < m - 1$ then
22                     $cpkt\_id = cpkt\_id + 1$
23                     $remain\_size = C$
24                 else
25                     return $partitions$
26                 end
27             end
28         end
29     end
30     return $partitions$
```

## 5.3 Checkpoint Partition Algorithm

GEMINI uses a checkpoint partition algorithm illustrated in Algorithm 2 to partition checkpoints for transmission pipelining. Given the set of profiled network idle timespans $\mathcal{T} = \{t_1, t_2, \ldots, t_d\}$ (discussed in Section 5.4), Algorithm 2 generates a scheduling of checkpoint partitions. Suppose there are $p$ GPU buffers in GEMINI and the size of each buffer is $R/p$, where $R$ is the total reserved GPU memory size. Suppose there are $m$ checkpoint replicas, and $m - 1$ replicas are sent to the remote CPU memory while one is stored locally. Suppose the time length of sending a partition of size $s$ to a receiver is $f(s) = \alpha + s/B$, where $\alpha$ is the startup time for transmission and $B$ is the network bandwidth [21, 72, 87].

Algorithm 2 uses a coefficient $\mu \in (0, 1)$ to consider the variance of the profiled timespans across iterations (Line 7). Because the size of each buffer is $R/p$, the maximum checkpoint chunk size is also $R/p$. The algorithm checks how many chunks it can insert in each idle timespan with multiple rounds. In each round, it compares $f(R/p)$ with the remaining idle timespan (*remain_span*). If *remain_span* is greater, it sets *size* to the maximum chunk size $R/p$ (Lines 9-10); otherwise, it sets the size to the amount of traffic volume that can be transmitted during *remain_span* (Line 11). It then compares *size* with the remaining checkpoint size (*remain_size*) and takes the smaller one as the chunk size (Line 14). It accordingly updates *remain_span* and *remain_size* for the next round (Lines 15-19). When *remain_size* equals zero, the algorithm finishes the partition of one checkpoint. If there are multiple checkpoint replicas for a higher failure recovery rate from CPU memory, the algorithm resets *remaining_size* as the checkpoint size and determines the partition for the new checkpoint again (Lines 21-23). The algorithm returns *partitions* after all the checkpoints are partitioned.

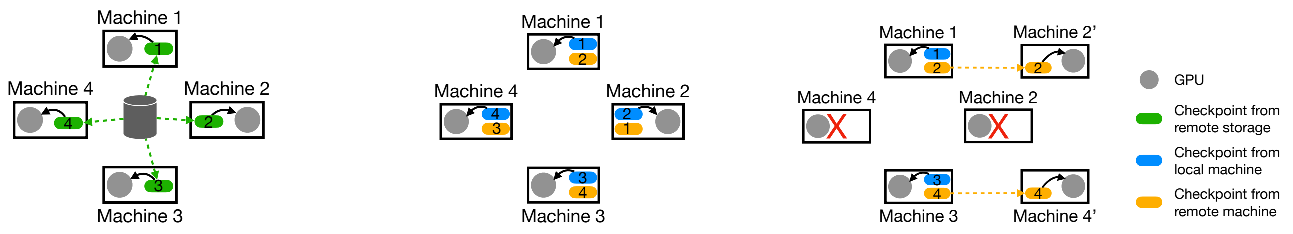
Our evaluation in Section 7 demonstrates that for all the evaluated large models, Algorithm 2 allows GEMINI to fully utilize the network idle timespans and enables it to perform checkpointing at the frequency of every iteration without interfering with training.

**Finish checkpointing within an iteration.** However, it is still possible that the total time required for checkpointing cannot be fit in the available network idle timespans. In such a scenario, GEMINI places the unfinished checkpoint traffic in the last idle timespan, as Algorithm 2 sets the interval of the last idle timespan as positive infinity (Line 2). Although checkpoint communications hinder the update operation and prolong the iteration time in this case, GEMINI can reduce the checkpoint frequency to amortize the incurred overhead.

**Move checkpoints from GPU to local CPU.** Each machine also needs to copy its checkpoint from GPU memory to its local CPU memory according to our placement strategy discussed in Section 4. This checkpoint copy incurs no traffic across machines. GEMINI also partitions this replica and overlaps its GPU-to-CPU copy with communications for training traffic. In this way, there is no interference between the local GPU-to-CPU copy of its own checkpoint and other checkpoints.

## 5.4 Online Profiling

GEMINI adopts online profiling for the first several iterations of training, e.g., 20 iterations in our implementation, without checkpointing in order to capture the network idle timespans during model training. It timestamps the start and the end time of all communication operations in an iteration to derive the timeline of communication traffic. GEMINI then obtains the average time interval of each idle timespan for subsequent checkpoint traffic scheduling. We observed that the profiled timeline remains almost constant across iterations, which is consistent with previous studies [77, 79, 86]. The normalized standard deviation of the measurements is


Machine 1
Machine 2
Machine 3
Machine 4
Machine 2'
Machine 4'
GPU
Checkpoint from remote storage
Checkpoint from local machine
Checkpoint from remote machine

**(a)** Existing solutions for any types of failures. All checkpoints are always retrieved from the remote persistent storage.

**(b)** Gemini for software failures. Checkpoints are at local and the retrieval time is negligible.

**(c)** Gemini for failures with two machines replaced. The newly added machines retrieve checkpoints from alive machines.

**Figure 6.** Illustrations of different mechanisms to recover training with four machines from different failures.

less than 10%. Gemini uses these idle timespan intervals to determine the checkpoint partitions in each idle timespan according to Algorithm 2 described in Section 5.3.

## 6 Resuming Training from Failures

Gemini achieves high-frequency checkpoints with the mixed checkpoint placement and the traffic interleaving algorithm. In this section, we will explain how Gemini uses the checkpoints to resume training when failures occur. We first define our failure classification and then describe how Gemini resumes training accordingly.

### 6.1 Failure Types

There are various failures that can occur during the training of large models [36, 55, 70, 75] and these failures have different root causes and consequences. We categorize these failures into two types from the perspective of recovery: software failures and hardware failures, following the literature [31, 36, 55, 70, 74, 75].

**Software failures** are caused by bugs in software or errors in data. Software failures can be fixed by restarting the training process without requiring hardware replacements.

**Hardware failures** are caused by hardware issues, such as GPU malfunctions and network failures. For example, bit corruptions induced by radiation can cause double bit error, leading to data corruptions [36, 75]. The network links and switches that connect GPU machines can fail [31, 71], disconnecting them from training. These failures can occur in a single machine or multiple machines simultaneously. The training cluster typically detects problematic machines and then replaces them with healthy ones before resuming training.

### 6.2 Failure Recovery Mechanisms

Existing checkpointing solutions [28, 65, 85] make no distinction between *software failures* and *hardware failures*. As shown in Figure 6a, they always retrieve the checkpoints from the remote persistent storage regardless of the failure type, resulting in costly wasted time. In this subsection, we will present the recovery mechanisms of Gemini for both types of failures, respectively.

**Software failures recovery.** Recovering from software failure does not require fetching checkpoints from other machines, and the training configurations (e.g., the rank ID of the machine) remain the same. When a software failure occurs, the training process is interrupted, but the hardware remains healthy and all checkpoints stored in CPU memory are still accessible. Because each machine stores a replica of its own checkpoint, all machines can directly recover training from their local checkpoints, as shown in Figure 6b.

**Hardware failures recovery.** When hardware failures occur, the training system needs to replace the failed machines. The root agent in Gemini interacts with the cloud operator (e.g., Auto Scaling Group platform in AWS) to replace the faulty machines with healthy ones. When recovering training from hardware failures, there are two cases: 1) there are still healthy machines in each checkpoint placement group assigned by Algorithm 1, and 2) there is at least one checkpoint placement group in which all machines fail simultaneously. We will next discuss these two cases.

**Case 1:** As each checkpoint placement group still has healthy machines maintaining checkpoint replicas, Gemini can fetch the checkpoint replica from them for newly added machines and then recover the training progress. Figure 6c illustrates an example with four machines and two machines, Machine 2 and Machine 4, just failed simultaneously. The root agent replaces the two failed machines with two healthy ones. The two newly added machines replace their positions, reuse their machine rank IDs, and retrieve their checkpoints from alive machines. Because a checkpoint replica of Machine 2 was stored in Machine 1, Machine 2′ (the one that replaced Machine 2) retrieves the checkpoint from Machine 1 for failure recovery. Machine 4′ also retrieves the checkpoint from Machine 4. The machines that have no failures can directly restart training from their local checkpoints.

**Case 2:** In this case, machines must retrieve checkpoints from the remote persistent storage to ensure all machines recover training consistently. Although part of the model checkpoints are still accessible in the alive GPU machines, they are not consistent with the ones in the remote persistent storage because they are stored from different iteration numbers. In practice, the majority of failures during large model training are software failures or hardware failures with one machine replaced; it is rare to have two or more machine failures at the same time [3, 14]. Even with multiple machine failures simultaneously, Gemini can still recover failures from CPU memory in most cases thanks to the checkpoint placement strategy, as we will discuss in Section 7.2.

**Standby machines.** In case of hardware failures, the cloud operator is expected to provide healthy machines to replace faulty ones immediately. However, this replacement operation heavily depends on the availability of healthy machines in the GPU cloud and it can take a non-deterministic duration to successfully reserve new machines for the current training workload. In order to minimize the waiting time resulting from machine replacement, the training cluster can pre-allocate a few standby machines. When a machine suffers from hardware failures, a standby machine can immediately become active to replace the failed one for failure recovery. After that, the root agent returns the failed one and requests another standby machine. Gemini allows users to specify different numbers of standby machines according to their training workloads and the availability of healthy machines in GPU clouds.

**Failure detection.** The cloud operators typically provide tools to detect training failures and locate the failed machines. For example, Amazon SageMaker [17] has tools for failure type detection and failure machine localization. Gemini relies on these tools to detect failures in large model training. In addition, the worker agents and the root agent in Gemini also periodically send heartbeat signals to the distributed key-value store for failure detection.

## 7 Evaluation

In this section, we will demonstrate the effectiveness of Gemini for failure recovery in large model training. Specifically, we will address the following research questions:

- **Failure recovery performance:** Can Gemini reduce the wasted time without harming training throughput? (Section 7.2)
- **Scalability:** How does Gemini perform under different failure frequencies and training scales? (Section 7.3)
- **Effectiveness of traffic interleaving:** How does our traffic interleaving algorithm affect the training throughput? (Section 7.4)

### 7.1 Implementation and Experimental Methodology

**Implementation.** We implement Gemini on top of DeepSpeed [63] and use ZeRO-3 setting [62]. we adopt etcd [9] as the distributed key-value store implementation to coordinate failure recovery. On the cloud provider side, we rely on Amazon EC2 Auto Scaling Groups (ASG) [4] to manage GPU machines. When failures are detected by ASG, the faulty machines are replaced with healthy ones. Such service is also available in Google Cloud [6] and Microsoft Azure [5]. Gemini reserves 128MB GPU memory for checkpoint communications. There are two CPU memory buffers to store the checkpoints: one for the completed checkpoint and the other for the ongoing one. When a failure occurs, the root agent notifies all alive agents to serialize the latest complete checkpoints with `torch.save()`, allowing PyTorch to load the saved checkpoints for failure recovery with `torch.load()`.

| Model size | Hidden size | Intermediate | #Layers | #AH |
|---|---|---|---|---|
| GPT-2 10B | 2560 | 10240 | 46 | 40 |
| GPT-2 20B | 5120 | 20480 | 64 | 40 |
| GPT-2 40B | 5120 | 20480 | 128 | 40 |
| RoBERTa 40B | 5120 | 20480 | 128 | 40 |
| BERT 40B | 5120 | 20480 | 128 | 40 |
| GPT-2 100B | 8192 | 32768 | 124 | 64 |
| RoBERTa 100B | 8192 | 32768 | 124 | 64 |
| BERT 100B | 8192 | 32768 | 124 | 64 |

**Table 2.** Configurations of different language models. AH is short for attention heads. GPT-2 10B means GPT with 10 billion parameters. The same naming convention applies to other models.

**Setups.** We conduct all experiments on AWS EC2 platform. Unless otherwise specified, we use 16 p4d.24xlarge instances for evaluations. Each instance has 1152GB CPU memory and it has 8 NVIDIA A100 (40GB) GPUs, which are interconnected via NVSwitch. p4d.24xlarge instances are connected through a 400Gbps elastic fabric adaptor (EFA) network. We adopt FSx [7] as the remote persistent storage and the aggregated bandwidth is 20Gbps. We also evaluate Gemini with p3dn.24xlarge instances, which have 8 NVIDIA V100 (32GB) GPUs and are connected to a 100Gbps EFA network. The used software versions are CUDA-11.6, DeepSpeed-v0.7.3, PyTorch-1.13, nccl-v2.14.3, and etcd-v3.5.

**Workloads.** We evaluate Gemini with popular and representative large deep learning models, including GPT-2 [61], BERT [27], and RoBERTa [43]. We vary the number of layers, hidden sizes, and intermediate sizes in these models [62, 87]. Table 2 summarizes the detailed model configurations. We use the sequence length 512 and the vocabulary size 50265 for the evaluation. We set the micro-batch size to 8 with mixed-precision and we enable the activation recomputation [39, 50] in the evaluation. The optimizer used is Adam [38]. The training dataset is Wikipedia-en corpus [46].

**Baselines.** We adopt two baselines, *Strawman* and *HighFreq*, for the evaluations. Strawman uses the checkpoint frequency following the setup in training BLOOM [3] and it checkpoints model states every three hours. HighFreq aims to fully saturate the bandwidth capacity of the remote persistent storage and it represents the best we can do with remote storage-based solutions. HighFreq first profiles both the checkpoint time $t_{ckpt}$ and the iteration time $T_{iter}$; it then checkpoints

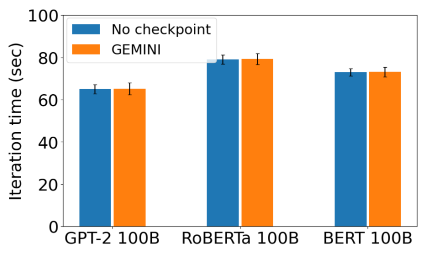


**Figure 7.** The iteration time of three large models without checkpoints and with Gemini.

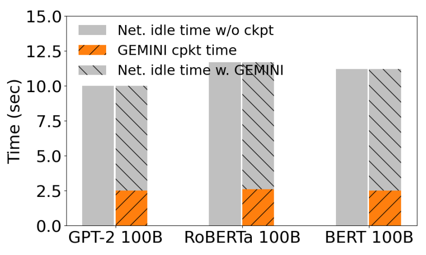


**Figure 8.** The network idle time of three large models without checkpoints and with Gemini.

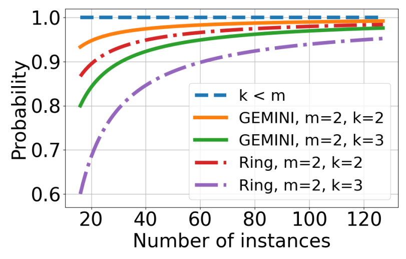


**Figure 9.** The probability that Gemini can recover failures from checkpoints stored in CPU memory.

the model states every $\lceil t_{ckpt}/T_{iter} \rceil$ iterations. Both baselines store the checkpoints in the remote persistent storage, while the difference is the checkpoint frequency. Note that Gemini also checkpoints to the remote persistent storage every three hours in addition to checkpointing to CPU memory.

## 7.2 Training Efficiency

In this subsection, we evaluate Gemini on both p4d.24xlarge and p3dn.24xlarge instances. We first use 16 p4d.24xlarge instances to demonstrate the performance advantages of Gemini over the baselines on large-scale model training. The largest model size we can train is 100B given the machine scale and the GPU memory size. Further increasing the model size causes GPU out-of-memory errors.

**Training time.** We examined Gemini's impacts on the training throughput by benchmarking GPT-2 100B, RoBERTa 100B, and BERT 100B. We carried out 50 training iterations with Gemini, which performed checkpointing for every iteration, and an equal number of iterations without checkpointing using vanilla DeepSpeed. Figure 7 shows the iteration times for both settings across the three models. We can find that Gemini does not affect the training iteration times. This is because the network idle time during training is adequate to accommodate the checkpoint traffic. Figure 8 confirms that there is still available network idle time even after Gemini inserts all the checkpoint traffic. It indicates that Gemini can achieve per iteration checkpointing without incurring extra overhead to the training throughput thanks to the traffic interleaving algorithm.

Since Gemini has negligible overhead for all the large models we evaluated, we use the GPT-2 100B model as the representative in the following part for brevity. RoBERTa and BERT have similar results and will not affect our conclusions.

**Wasted time.** We next evaluate the wasted time when a failure occurs. We first analyze the probability that Gemini can recover failures from CPU memory. Given the checkpoint replica number $m$, the probability is determined by the number of instances $k$ that need to be replaced simultaneously (failures occurred on those instances). When $k < m$, Gemini can always recover training from CPU memory. When $k \geq m$, we can calculate the probability according to Corollary 1. Figure 9 plots the probability that Gemini can recover failures from CPU memory under different settings. The probability increases with the number of instances $N$. Suppose there are two checkpoint replicas, i.e., $m = 2$. When $N = 16$ and $k = 2$, Gemini has a probability of 93.3%; when $k = 3$, it still has a probability of 80.0%. We also consider the ring strategy, in which instance $i$ stores its model states in itself and instance $(i + 1)$ mod $N$. When $N = 16$ and $k = 3$, Ring's probability is 25.0% lower than that of Gemini. According to OPT-175B [85] observation, there are 1.5% instances that fail every day. Even for a thousand-scale training cluster, the possibility of two instances having failures at the same time is very limited. Therefore, Gemini with $m = 2$ can recover failures from CPU memory for most cases.

We next calculate the average wasted time based on the measured iteration time, checkpoint time, and retrieval time according to Expression (1). Figure 10 shows the average wasted time for training of GPT-2 100B on 16 p4d.24xlarge instances with different numbers of replaced instances. The average wasted time of both Strawman and HighFreq is deterministic because the checkpoints are always retrieved from the remote persistent storage when failures occur. In contrast, the average wasted time of Gemini varies. When there is no instance replaced, e.g., due to software failures, the checkpoints are already at the local CPU memory. The average wasted time in this case is 1.5× the iteration time ($1.5T_{iter}$). When there is only one instance replaced or two instances are replaced but training can be recovered from the CPU memory, the extra overhead for failure recovery is to retrieve checkpoints from other instances and the retrieval time is less than three seconds. In these cases, Gemini can reduce the average wasted time by more than 13× compared to HighFreq. However, when two instances are replaced and training cannot be recovered from the CPU memory, of which the possibility is 6.7% with 16 instances according to Figure 9, Gemini degrades to Strawman.

**Checkpoint time.** To showcase the advantage of Gemini in terms of checkpoint time, Figure 11 displays the checkpoint time reduction of Gemini over the baselines under

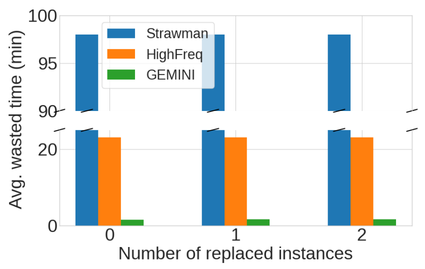


**Figure 10.** The average wasted time of GPT-2 100B with different numbers of replaced instances.

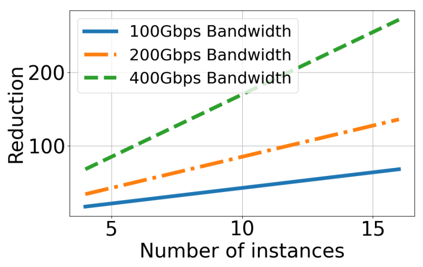


**Figure 11.** The checkpoint time reduction of Gemini over the baselines under different network bandwidth.

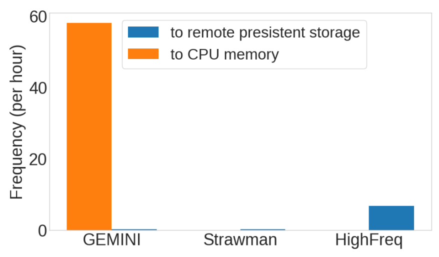
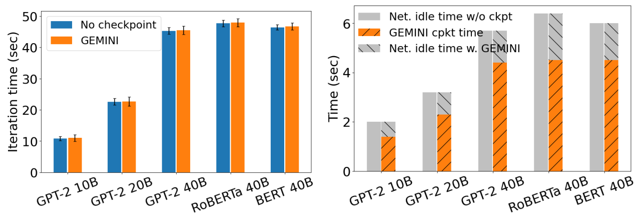


**Figure 12.** Gemini achieves a much higher checkpoint frequency than the two baselines.



**(a)** The iteration time. **(b)** The network idle time.

**Figure 13.** Gemini is generalized to p3dn.24xlarge instances and other models.

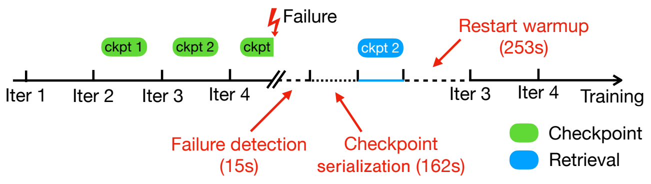


**Figure 14.** The overhead of failure recovery for GPT-2 100B training with Gemini. A failure occurs during Iteration 4 and one instance is replaced.

different network bandwidths and different numbers of instances. Both baselines, Strawman and HighFreq, have the same checkpoint time and it stays almost the same as the number of machines increases from 4 to 16 because the aggregated bandwidth of the remote persistent storage is fixed. In contrast, Gemini's checkpoint time reduces with an increase in the number of instances in our testbed because it utilizes the aggregated network bandwidth among GPU machines to write checkpoints to the CPU memory. The checkpoint time reduction also increases with the network bandwidth connecting GPU instances. For example, with 16 p4d.24xlarge instances, the reduction is 65× with a 100Gbps network, and it increases to more than 250× with a 400Gbps network. It is very challenging for remote persistent storage to achieve comparable performance as Gemini. To match the checkpoint time of Gemini in our scenario, which involves 16 instances, persistent storage would need to achieve an aggregated bandwidth of 6.4Tbps theoretically.

**Checkpoint frequency.** Gemini checkpoints model states to CPU memory for every iteration. The iteration time of GPT-2 100B with 16 p4d.24xlarge is 62 seconds, but the checkpoint time with Gemini is less than 3 seconds. As shown in Figure 12, Gemini improves the checkpoint frequency over HighFreq by 8× and over Strawman by more than 170×. Note that the checkpoint frequency of Gemini is bounded by the iteration time and it can achieve an even higher frequency with the computation advancement of accelerators.

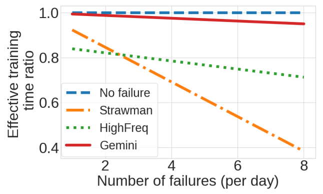
We then demonstrate that Gemini can also efficiently support other training models on p3dn.24xlarge instances. The largest model size we can train with this hardware setting is 40B. Further increasing the model size causes GPU out-of-memory errors in our testbed.

**Model training on p3dn.24xlarge.** Figure 13a illustrates that Gemini minimally affects the training throughput using 16 p3dn.24xlarge instances across various model sizes (10B, 20B, and 40B) and model architectures (GPT-2, RoBERTa, and BERT). The training efficiency aligns with the findings from 16 p4d.24xlarge instances. Figure 13b contrasts network idle times during model training without checkpoints and with Gemini, revealing that the network idle time is still sufficient to accommodate the checkpoint traffic.

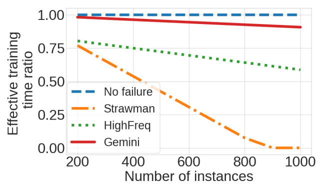
## 7.3 System Scalability

In this subsection, we first report the overheads incurred by failures in Gemini and the baselines. We then use simulation to demonstrate that Gemini is scalable to scenarios with frequent failures and to support LLM training with thousands of instances.

**Overheads incurred by failures.** Besides the lost training progress, the checkpoint time, and the retrieval time, there are other overheads in Gemini to recover training from a failure. We train GPT-2 100B on 16 p4d.24xlarge instances and the training process is illustrated in Figure 14. Gemini checkpoints the model states to the CPU memory for every iteration. An instance failure is triggered during Iteration 4 and it takes 15 seconds for the root agent to detect this failure.



**(a)** Different failure rates.



**(b)** Different instance numbers.

**Figure 15.** The scalability of Gemini under simulation.

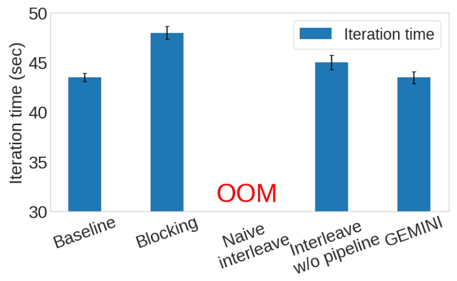


**Figure 16.** The iteration time of GPT-2 40B with different schemes for checkpointing to CPU memory. OOM is short for out of memory.

The root agent then notifies all alive instances to serialize the checkpoints stored in CPU memory with `torch.save()`. We observe that this operation is time-consuming and it takes 162 seconds to finish the serialization of two checkpoint replicas, one is from local and the other is from another instance. We also measure the waiting time to successfully reserve a new p4d.24xlarge instance with ASG to estimate the extra instance-replacing overhead in case of hardware failures, which is around 4-7 minutes. Another noticeable overhead is the restart warmup time and it takes more than four minutes before the training can proceed from Iteration 3. To sum up, in our testbed, the total overhead resulting from a failure that can be recovered from CPU memory is around 7 minutes for software failures and 12 minutes for hardware failures. Note that the instance-replacing overhead for hardware failures can be greatly reduced by standby machines.

Although the two baselines have no checkpoint serialization overhead when a failure occurs, they have such overhead for every checkpoint to the remote persistent storage. Their checkpoint communications to the remote persistent storage are asynchronous to computation, but they need to serialize the checkpoints with `torch.save()`, which blocks training. HighFreq checkpoints the model states every nine iterations and the incurred overhead for each checkpoint serialization is around 81 seconds. Strawman also has this overhead, but it is negligible due to the low frequency.

Based on the incurred overhead by one failure, we can simulate the training performance of GPT-2 100B with different failure rates and different numbers of instances. We consider software failures in the simulation because recovering training from hardware failures has a similar overhead as from software failures if standby machines are used.

**Scaling to frequent failures.** To evaluate the impact of failure rates, we conducted simulations of training performance using 16 p4d.24xlarge instances and different checkpointing solutions. We measured the training performance using a metric called the effective training time ratio, which indicates the percentage of productive training progress achieved in a given period of time. Failures decrease this ratio due to the overheads for failure recovery. The effective training time ratios with different solutions are shown in Figure 15a. We found that even with 8 failures per day, Gemini remains highly efficient with a performance ratio close to the baseline with no failures. However, the costly overhead of checkpoint serialization, i.e. invoking `torch.save()`, in HighFreq significantly hurts its performance. Even without any failures, 14.5% time is spent on checkpoint serialization. On the other hand, Gemini only serializes checkpoints when failures occur. Strawman is worse than HighFreq due to its prohibitive wasted time.

**Scaling to more instances.** We also simulate the training performance with different numbers of instances involved in training. Following the training report of OPT-175B [85], we assume that 1.5% instances fail every day. The failure frequency increases with the number of instances. Figure 15b shows that with 1000 instances, the effective training time ratio of Gemini is still around 91%, which is 54% higher than HighFreq. Training with Strawman for failure recovery can hardly proceed because of the frequent failures and the prohibitive wasted time.

## 7.4 Effectiveness of Traffic Interleaving

In this subsection, we evaluate the effectiveness of Gemini's traffic interleaving algorithm. To understand the performance contributions of its two approaches, we report the iteration time of GPT-2 40B on 16 p3dn.24xlarge instances with the following schemes for checkpointing to CPU memory.

- **Baseline.** It is the model training without checkpointing.
- **Blocking.** It checkpoints the model states to CPU memory, but the checkpoint traffic blocks training traffic at the beginning of each iteration.
- **Naïve interleave.** It partitions checkpoint traffic for interleaving, but each network idle timespan only has one checkpoint partition.
- **Interleave without pipeline.** Each idle timespan can have multiple partitions, but it only uses one GPU buffer for checkpoint communications. The buffer size is 128MB.
- **Gemini.** It uses four small sub-buffers for pipelining checkpoint communications and the size of each buffer is 32MB.

As shown in Figure 16, the iteration time with Blocking is 10.1% higher than the Baseline due to the extra checkpoint time. Naïve interleave can cause GPU out-of-memory (OOM) errors because it requires a large GPU memory buffer for checkpoint communications. For example, the largest idle time span profiled during training is 1.6s and the required memory buffer size is more than 2GB on each GPU. Interleave without pipeline can greatly reduce the required GPU memory buffer size and avoid OOM error, but communications have to wait for GPU-to-CPU copy. The total network idle time becomes insufficient to accommodate the checkpoint traffic in this case and it worsens the iteration time by 3.5%. In contrast, the iteration time with Gemini is almost the same as the Baseline because it can fully utilize the network idle time by pipelining checkpoint communications.

## 8 Related Work

**Checkpointing in deep learning.** Deep learning frameworks, such as PyTorch [57], TensorFlow [18], and MXNet [22], provide users with the interfaces to checkpoint model states during training for failure recovery. Unlike Gemini, it is the users' responsibility to decide how to checkpoint, such as the checkpoint frequency and storage location. To reduce checkpointing overheads, DeepFreeze [51] performs asynchronous checkpointing but stores checkpoints in remote persistent storage. CheckFreq [48] dynamically adjusts the checkpointing frequency, but the remote storage bandwidth limits the highest frequency. In contrast, Gemini stores checkpoints in CPU memory, enabling much higher frequencies than DeepFreeze and CheckFreq. Check-N-Run [28] compresses checkpoints with lossy schemes to reduce required storage, but this may harm model accuracy and incur compression overheads. Gemini stores the original checkpoints without impacting accuracy or incurring compression overheads. Gandiva [82] assumes healthy machines for checkpointing with an on-demand checkpoint mechanism for job migration. Because any machine involved in training can experience hardware failures, Gandiva's checkpoint mechanism cannot handle this case in which checkpoints stored in failed machines will get lost. Furthermore, its on-demand checkpointing cannot tackle unexpected failures during large model training. In contrast, Gemini aims to recover training from both unexpected software and hardware failures.
**Checkpoint and data placement in distributed systems.** Diskless checkpointing [59] stores checkpoints in CPU memory. It requires processors to encode a checkpoint with parity and their checkpoints can be recalculated when a processor fails. However, encoding and decoding a checkpoint of large model training is extremely expensive. Instead, Gemini employs redundant checkpoints for failure recovery. FTC-Charm++ [88] stores two checkpoint copies on two processors for fault tolerance. However, it lacks an analysis of optimal checkpoint placements. Some distributed systems are proposed for data placement in clouds [19, 25, 80]. For example, CRUSH [80] distributes data replicas uniformly among machines to maintain a statistically balanced utilization of storage and bandwidth resources; Volley [19] develops automated techniques to place application data across data centers. In contrast, Gemini groups machines involved in training for checkpoint placement to maximize the probability of failure recovery from checkpoints stored in CPU memory. Unlike traditional distributed systems for checkpoint and data placement, a key challenge in Gemini is to schedule checkpoint traffic to minimize interference with training, which differentiates Gemini from existing work.
**Communication scheduling in distributed training.** ByteScheduler [58], TicTac [32], and P3 [35] aim to improve the performance of training by scheduling the communication orders of tensors. These works primarily focus on accelerating training communication. They are orthogonal and complementary to Gemini because Gemini focuses on minimizing interference with training communication by scheduling checkpoint communications.
**Failure recovery with spot instances.** Bamboo [73] uses redundant computation to provide resilience and fast recovery for training large DNN models on preemptible instances. Gemini checkpoints to CPU memory and doesn't require redundant computation. Varuna [20] also enables large model training on preemptible instances, but it requires users to manage the checkpoints, such as the frequency and the storage, for failure recovery. In contrast, Gemini offers transparent checkpointing for failure recovery, eliminating the need for users to manage checkpoints.

## 9 Conclusion and Future Work

This paper presents Gemini, a distributed training system that enables fast failure recovery for large model training. By checkpointing to CPU memory, Gemini achieves high checkpoint frequencies to minimize wasted time and incurs no overhead on training throughput for large model training. Experiments on GPU clusters show that Gemini achieves more than 13× faster failure recovery compared to existing solutions, without hurting the training throughput. While the current implementation of Gemini is built upon ZeRO-3, we believe the proposed design is applicable to other parallelisms, such as pipeline parallelism [34, 49], tensor parallelism [67], data parallelism [37, 66, 87], and a combination of them [50, 89], which is part of our future work. In addition, we plan to apply Gemini to the training system with other accelerators such as AWS Trainium.

## Acknowledgment

We would like to thank our shepherd Lidong Zhou and the anonymous reviewers for providing valuable feedback. Zhuang Wang and T. S. Eugene Ng are partially supported by the NSF under CNS-2214272 and CNS-1815525.

## References

[1] NVIDIA NCCL. https://developer.nvidia.com/NCCL, 2021.

[2] AWS Trainium. https://aws.amazon.com/machine-learning/trainium/, 2022.

[3] BLOOM Chronicles. https://github.com/bigscience-workshop/bigscience/blob/master/train/tr11-176B-ml/chronicles.md, 2022.

[4] Auto Scaling in AWS. https://docs.aws.amazon.com/autoscaling/, 2023.

[5] Auto Scaling in Azure. https://learn.microsoft.com/en-us/azure/app-service/manage-scale-up, 2023.

[6] Auto Scaling in Google Cloud. https://cloud.google.com/compute/docs/autoscaler, 2023.

[7] AWS FSx. https://aws.amazon.com/fsx/, 2023.

[8] ChatGPT. https://openai.com/blog/chatgpt, 2023.

[9] etcd. https://etcd.io/, 2023.

[10] GPU instances in Google Cloud Platform. https://cloud.google.com/compute/docs/gpus, 2023.

[11] ND40rs_v2 in Azure. https://learn.microsoft.com/en-us/azure/virtual-machines/ndv2-series, 2023.

[12] ND96asr_v4 in Azure. https://learn.microsoft.com/en-us/azure/virtual-machines/nda100-v4-series, 2023.

[13] NVIDIA DGX A100. https://www.nvidia.com/en-gb/data-center/dgx-a100/, 2023.

[14] OPT-175B logbook. https://github.com/facebookresearch/metaseq/tree/main/projects/OPT/chronicles, 2023.

[15] P3dn.24xlarge in AWS. https://aws.amazon.com/ec2/instance-types/p3/, 2023.

[16] P4d.24xlarge in AWS. https://aws.amazon.com/ec2/instance-types/p4/, 2023.

[17] SageMaker. https://docs.aws.amazon.com/sagemaker/index.html, 2023.

[18] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016.

[19] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Habinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, 2010.

[20] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[21] Ammar Ahmad Awan, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. Optimized broadcast for deep learning workloads on dense-GPU InfiniBand clusters: MPI or NCCL? In *Proceedings of the 25th European MPI Users' Group Meeting*, pages 1–9, 2018.

[22] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[23] Yu Chen, Zhenming Liu, Bin Ren, and Xin Jin. On efficient constructions of checkpoints. In *International Conference on Machine Learning*, pages 1627–1636. PMLR, 2020.

[24] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[25] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 37–48, 2013.

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

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

[28] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: a checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, 2022.

[29] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 25–36, 2012.

[30] Roxana Geambasu, Amit A Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M Levy. Comet: An active distributed key-value store. In *OSDI*, pages 323–336, 2010.

[31] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 350–361, 2011.

[32] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. TicTac: Accelerating distributed deep learning with communication scheduling. *Proceedings of Machine Learning and Systems*, 1:418–430, 2019.

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

[34] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. GPipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[35] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based parameter propagation for distributed DNN training. *Proceedings of Machine Learning and Systems*, 1:132–145, 2019.

[36] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *USENIX Annual Technical Conference*, pages 947–960, 2019.

[37] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 463–479, 2020.

[38] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[39] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *arXiv preprint arXiv:2205.05198*, 2022.

[40] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[41] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch, and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2019.

[42] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel

training. *Proceedings of the VLDB Endowment*, 13(12).

[43] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[44] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, et al. Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. *Proceedings of Machine Learning and Systems*, 3:637–651, 2021.

[45] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pages 937–954, 2020.

[46] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[47] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[48] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. Checkfreq: Frequent, fine-grained DNN checkpointing. In *FAST*, volume 21, pages 203–216, 2021.

[49] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[50] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[51] Bogdan Nicolae, Jiali Li, Justin M Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 172–181. IEEE, 2020.

[52] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference*, pages 305–319, 2014.

[53] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[54] Andrew Or, Haoyu Zhang, and Michael Freedman. Resource elasticity in distributed deep learning. *Proceedings of Machine Learning and Systems*, 2:400–411, 2020.

[55] George Ostrouchov, Don Maxwell, Rizwan A Ashraf, Christian Engelmann, Mallikarjun Shankar, and James H Rogers. GPU lifetimes on Titan supercomputer: Survival analysis and reliability. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[56] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

[57] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[58] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 16–29, 2019.

[59] James S Plank, Kai Li, and Michael A Puening. Diskless checkpointing. *IEEE Transactions on parallel and Distributed Systems*, 9(10):972–986, 1998.

[60] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K Panda. Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs. In *2013 42nd International Conference on Parallel Processing*, pages 80–89. IEEE, 2013.

[61] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[62] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[63] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[64] Davide Rossetti and S Team. GPUDirect: Integrating the GPU with a network interface. In *GPU Technology Conference*, page 185, 2015.

[65] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. BLOOM: A 176B-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[66] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[67] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[68] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using DeepSpeed and Megatron to train Megatron-Turing NLG 530B, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

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

[70] Amir Taherin, Tirthak Patel, Giorgis Georgakoudis, Ignacio Laguna, and Devesh Tiwari. Examining failures and repairs on supercomputers with multi-GPU compute nodes. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 305–313. IEEE, 2021.

[71] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. NetBouncer: Active device and link failure localization in data center networks. In *NSDI*, pages 599–614, 2019.

[72] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1), 2005.

[73] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *NSDI*, 2023.

[74] Devesh Tiwari, Saurabh Gupta, George Gallarno, Jim Rogers, and Don Maxwell. Reliability lessons learned from GPU experience with the Titan supercomputer at oak ridge leadership computing facility.

In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 1–12, 2015.

[75] Devesh Tiwari, Saurabh Gupta, James Rogers, Don Maxwell, Paolo Rech, Sudharshan Vazhkudai, Daniel Oliveira, Dave Londo, Nathan DeBardeleben, Philippe Navaux, et al. Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–342. IEEE, 2015.

[76] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[77] Zhuang Wang, Haibin Lin, Yibo Zhu, and TS Eugene Ng. Hi-speed DNN training with espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys 23)*, pages 867–882, 2023.

[78] Zhuang Wang, Xinyu Wu, Zhaozhuo Xu, and T. S. Eugene Ng. Cupcake: A compression scheduler for scalable communication-efficient distributed training. *Proceedings of Machine Learning and Systems*, 5, 2023.

[79] Zhuang Wang, Zhaozhuo Xu, Xinyu Wu, Anshumali Shrivastava, and T. S. Eugene Ng. DRAGONN: Distributed randomized approximate gradients of neural networks. In *International Conference on Machine Learning*, pages 23274–23291. PMLR, 2022.

[80] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 122–es, 2006.

[81] Yidi Wu, Kaihao Ma, Xiao Yan, Zhi Liu, Zhenkun Cai, Yuzhen Huang, James Cheng, Han Yuan, and Fan Yu. Elastic deep learning in multi-tenant GPU clusters. *IEEE Transactions on Parallel and Distributed Systems*, 33(1):144–158, 2021.

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

[83] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. PipeMare: Asynchronous pipeline parallel DNN training. *Proceedings of Machine Learning and Systems*, 3:269–296, 2021.

[84] Shanshan Zhang, Ce Zhang, Zhao You, Rong Zheng, and Bo Xu. Asynchronous stochastic gradient descent for DNN training. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6660–6663. IEEE, 2013.

[85] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[86] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML*, pages 8–13, 2020.

[87] Zhen Zhang, Shuai Zheng, Yida Wang, Justin Chiu, George Karypis, Trishul Chilimbi, Mu Li, and Xin Jin. MiCS: Near-linear scaling for training gigantic model on public. *Proceedings of the VLDB Endowment*, 16(1):37–50, 2022.

[88] Gengbin Zheng, Lixia Shi, and Laxmikant V Kalé. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *2004 ieee international conference on cluster computing (ieee cat. no. 04EX935)*, pages 93–103. IEEE, 2004.

[89] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association.

## Appendix for Gemini

Appendices have not been peer-reviewed.

### A Proof of Theorem 1

**Theorem 1.** *To address Problem 1 for checkpoint placement:*

1. *When N is divisible by m, the mixed placement strategy (equals group placement strategy) is the optimal placement strategy.*
2. *When N is not divisible by m, the mixed placement strategy minimizes the checkpoint communication time. Its failure recovery probability from CPU memory is near-optimal and the gap is bounded by* $(2m-3)/\binom{N}{m}$.

*Proof.* We first introduce two observations for checkpoint placements. (1) The optimal strategy requires $m$ machines to store the $m$ checkpoint copies of each machine to maximize the recovery probability. If there are only $m'$ machines to store the $m$ copies, where $m' < m$, it is equivalent to the strategy with only $m'$ copies for recovering failures from CPU memory. (2) The optimal strategy requires Machine $i$ to store one copy of its own machine checkpoint to minimize the checkpointing time. If so, each machine only needs to send out $m-1$ checkpoint copies. Otherwise, it has to send out $m$ copies which leads to a higher checkpointing time.

The checkpoint communication time with the Group strategy is minimized because each machine sends out and receives $m-1$ checkpoint copies, no matter whether $N$ is divisible by $m$ or not. We next analyze the probability that Gemini can recover failures from CPU memory.

Assume there are $k$ machines disconnected at the same time. Gemini can certainly recover failures from CPU memory when $k < m$ because there are $m$ copies in $m$ instances. We mainly discuss the case that $k = m$ here because the failure rate with $k+1$ machines disconnected simultaneously is much lower than that with $k$ machines disconnected simultaneously in practice.

For Machine $i$, we denote the set of machines that store its machine checkpoints as $s_i$ and it has $\binom{N}{m}$ possible combinations. Then a strategy can be expressed as $\mathcal{S} = \{s_1, s_2, \ldots, s_N\}$. Because it is possible that $s_i = s_j$ when $i \neq j$, we define $\mathcal{S}' = unique(\mathcal{S})$ and $n = |\mathcal{S}'|$. The union of these $n$ sets in $\mathcal{S}'$ covers all the $N$ machines because each machine stores a local checkpoint.

We denote the set of the $m$ disconnected machines as $s_d$. Note that $k = m$ in our analysis. Gemini cannot recover training from CPU memory when $s_d$ is an element in $\mathcal{S}'$. If so, all the $m$ copies of a machine checkpoint get lost and the model checkpoints stored in CPU memory become incomplete and invalid for failure recovery. The probability that $s_d$ is an element in $\mathcal{S}'$ is $n/\binom{N}{m}$, which linearly increases with $n$.

**Probability upper bound.** The upper bound of the probability is $1 - \lceil \frac{N}{m} \rceil / \binom{N}{m}$ because $n \geq N/m$. If $\lceil n < N/m \rceil$, the size of the union of the $n$ sets is at most $nm < N$. It contradicts the requirement that they must cover the $N$ machines.

**When $N$ is divisible by $m$.** Group placement strategy can achieve the upper bound. Machines in the same group have the same set of machines to store their checkpoints. Because there are $N/m$ groups, the number of unique sets in $\mathcal{S}$ is $N/m$. The probability is then $\mathcal{S}'$ is $\lceil \frac{N}{m} \rceil / \binom{N}{m}$, which is the lower bound. Therefore, we can conclude Group placement strategy is optimal for Problem 1 when $N$ is divisible by $m$.

**When $N$ is not divisible by $m$.** For the first $\lfloor N/m \rfloor - 1$ groups, machines in the same group have the same set of machines to store their checkpoints. For the last group, each machine has a distinct set of machines to store its checkpoints and there are $N - m(\lfloor N/m \rfloor - 1)$ unique sets. Therefore, the total number of unique sets in $\mathcal{S}$ is $N - (m-1)(\lfloor N/m \rfloor - 1)$. The gap between the upper bound and probability with the mixed placement strategy is bounded by $(2m-3)/\binom{N}{m}$. Since $N \gg m$ and $m$ is practically very small, the probability is very close to the upper bound. □

### B Proof of Corollary 1

**Corollary 1.** *When N is divisible by m and k machines are disconnected simultaneously, the probability that Gemini can recover failures from CPU memory is*

$$\begin{cases} \Pr(N,m,k) = 1, & \text{if } k < m \\ \Pr(N,m,k) \geq \max\{0, 1 - \frac{N\binom{N-m}{k-m}}{m\binom{N}{k}}\}, & \text{if } m \leq k \leq N \end{cases} \tag{4}$$

*Proof.* Gemini can certainly recover failures when $k < m$ because there are available checkpoint replicas in at least one of the machines. We then consider $m \leq k \leq N$.

With Algorithm 1 there are $N/m$ groups in $\mathcal{G}$ after the group placement strategy. When $k$ machines fail at the same time, if there exist $m$ failed machines forming a group that is an element of $\mathcal{G}$, it indicates that the checkpoints stored in CPU memory become incomplete and training has to recover from the remote persistent storage.

We first consider the case $m \leq k < 2m$. The number of combinations to choose $k$ machines from $N$ machines is $\binom{N}{k}$. The number of combinations causing incomplete checkpoints in CPU memory is $\frac{N}{m}\binom{N-m}{k-m}$. Therefore, the probability that Gemini can recover failures from CPU memory is

$$Pr(N,m,k) = 1 - \frac{N\binom{N-m}{k-m}}{m\binom{N}{k}}, \text{if m} \leq \text{k} < \text{2m}. \tag{5}$$

We then consider the case $k \geq 2m$. When we use the same method for $m \leq k < 2m$ to count the number of combinations, some combinations are counted more than once and the total number of combinations is less than $\frac{N}{m}\binom{N-m}{k-m}$. Therefore, the probability probability that Gemini can recover failures from CPU memory is

$$\Pr(N,m,k) > \max\{0, 1 - \frac{N\binom{N-m}{k-m}}{m\binom{N}{k}}\}, \text{if } k \geq m. \tag{6}$$

We then have Corollary 1 by combining the two cases together. □