Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs

John Thorpe[†♣] Pengzhan Zhao[†♣] Jonathan Eyolfson[†] Yifan Qiao[†] Zhihao Jia[‡]
Minjia Zhang[§] Ravi Netravali[*] Guoqing Harry Xu[†]
UCLA[†] *CMU*[‡] *Microsoft Research*[§] *Princeton University*[*]

Abstract

DNN models across many domains continue to grow in size, resulting in high resource requirements for effective training, and unpalatable (and often unaffordable) costs for organizations and research labs across scales. This paper aims to significantly reduce training costs with effective use of preemptible instances, i.e., those that can be obtained at a much cheaper price while idle, but may be preempted whenever requested by priority users. Doing so, however, requires new forms of *resiliency* and *efficiency* to cope with the possibility of frequent preemptions – a failure model that is drastically different from the occasional failures in normal cluster settings that existing checkpointing techniques target.

We present Bamboo, a distributed system that tackles these challenges by introducing *redundant computations* into the training pipeline, *i.e.*, whereby one node performs computations over not only its own layers but also over some layers in its neighbor. Our key insight is that training large models often requires *pipeline parallelism* where "pipeline bubbles" naturally exist. Bamboo carefully fills redundant computations into these bubbles, providing resilience at a low cost. Across a variety of widely used DNN models, Bamboo outperforms traditional checkpointing by **3.7**× in training throughput, and reduces costs by **2.4**× compared to a setting where on-demand instances are used.

1 Introduction

DNNs are becoming progressively larger to deliver improved predictive performance across a variety of tasks, including computer vision and natural language processing. For instance, recent language models such as BERT [66] and GPT [50] already have a massive number of parameters, and their newer variants continue to grow at a rapid pace. For example, BERT-large has 340 million parameters, GPT-2 has 1.5 billion, and GPT-3 increases to 175 billion; the next generation of models embed upwards of trillions of parameters [17].

Of course, model growth also entails larger training costs. For instance, GPT-3 consumes several thousand petaflop/s-days, costing over $12 million to train on a public cloud (needing hundreds of GPU servers) [6]. Unfortunately, such costs are prohibitive for small organizations. Even for large tech firms, training today's models incurs an exceedingly high monetary cost that eventually gets billed to the training department. While pretrained models may be reused and fine-tuned for different applications, training new models is often required to keep pace with changing or emerging workloads and datasets.

Although there exists a body of work on improving the training of large models [38, 39, 26, 9, 7, 11, 12, 18, 54, 53, 64, 72, 24, 28, 31], existing techniques focus primarily on *scalability* and *efficiency*, with monetary costs often being neglected. However, when *affordability* and *accessibility* are considered, resource usage becomes a key concern and none of these techniques were targeted at improving *cost-efficiency* (*e.g.*, performance-per-dollar) for training.

Preemptible Instances. This paper explores the possibility of using preemptible instances—a popular class of cheap cloud resources—to reduce the cost of training large models. There are several kinds of preemptible instances. For example, major public clouds provide *spot instances* with a price much cheaper than *on-demand instances*—*e.g.*, the hourly rate of a GPU-based spot instance is only ∼30% of that for its on-demand counterpart on Amazon EC2 [3]. As another example, large datacenters often maintain certain amounts of compute resources that can be allocated for any non-urgent tasks but will be preempted as urgent tasks arise [41, 5]. Similarly, recent ML systems [27, 69, 4] allow training jobs to use inference-dedicated machines to fully utilize GPU resources but preempts those machines when high-priority inference jobs arrive. The presentation of this paper focuses on spot instances, but we note that our techniques are generally applicable to any type of preemptible resources.

Despite their substantial cost benefits, preemptible instances pose major challenges in reliability and efficiency due the frequent and unpredictable nature of their preemptions. When and how many instances get preempted depends primarily on the number of priority jobs/users in a cluster. In a public spot market, preemption can also result from the market price exceeding the user's bid price. While price-based preemption can be avoided via a high bid price (*e.g.*, the on-demand price), capacity-based preemption is unavoidable. Preemption patterns vary drastically across clouds and even across families/zones on the same cloud (§3).

Given the unpredictable nature of spot instances, users can often only run short, stateless jobs and simply restart these jobs if they get preempted. Model training, on the contrary, is stateful and time-consuming. Discarding the state (*e.g.*, learned weights) upon each instance preemption not only

♣ Contributed equally.

wastes computation but also prevents training from making progress. Checkpointing-based techniques can reduce wasted computation to a degree, but still spend a significant fraction of the training time (*e.g.*, **77%** when training GPT-2 with 64 EC2 spot instances, see §3) on restarting and redoing prior work in the presence of frequent preemptions [20, 21]—a largely different scenario compared to conventional clusters where failures are rare.

Bamboo. This paper presents Bamboo, a distributed system that provides resilience and efficiency for DNN training over preemptible instances. Bamboo supports both pipeline parallelism and (pure) data parallelism with the same approach. Since pipeline parallelism is a more complex and general approach (for training large models), our discussion focuses on pipeline parallelism; we briefly discuss our support for pure data parallelism in §B. Bamboo currently does *not* support model parallelism.

Redundant Computation. Key to the success of Bamboo is a set of novel techniques centered around *redundant computation* (RC), inspired by how disk redundancies such as RAID [45] provide resilience in the presence of disk failures. A training system that uses pipeline parallelism runs a set of data-parallel pipelines, each training on a partition of the dataset. Each node[1] in a data-parallel pipeline performs (forward and backward) computations over a shard of NN layers with a *microbatch* of data items [24]. Bamboo lets each node in each data-parallel pipeline carry its own shard of layers as well as its successor's shard. Each node performs *normal* computation over its own layers and *redundant* computation over its successor's layers. The reason why we use a neighbor node (as opposed to a random node) to run RC is to exploit data locality in pipeline parallelism (see §5). Upon a node preemption, its predecessor has all the information (*e.g.*, layers, activations) needed for the training to progress; continuing training requires running a failover schedule on the predecessor node without wasting prior computations.

At first glance, running RC on every node appears infeasible due to concerns with both time and memory. Bamboo overcomes these challenges by taking into account pipeline characteristics to carefully reduce/hide these overheads.

First, to minimize the time overhead from RC, Bamboo leverages a key insight that *bubbles* [24, 51] inherently exist in systems using synchronous pipeline parallelism (§2). Bubbles are idle times on each node due to the gaps between the forward and backward processing of microbatches (Figure 1). Bamboo schedules the forward redundant computation (FRC) on each node asynchronously into the bubble. FRC entails a node doing the forward pass over its successor's layers using the output of its own active layers and is the main way Bamboo achieves redundancy. For the part of FRC that cannot fit into the bubble, Bamboo overlaps it with the normal computation. As a result, FRC incurs a tolerable overhead (*i.e.*, no extra communication is needed due to locality, and it can overlap with normal computation), and hence Bamboo performs it *eagerly* in each epoch. If a node is preempted during a forward pass, the pipeline continues after a node rerouting step whose overhead is negligible.

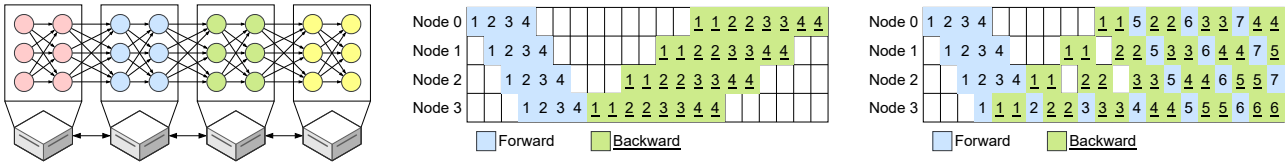
In addition to FRC, the system must find a way to generate the redundant version of the intermediate data related to backwards passes for the successor node. This can be accomplished by using the backward redundant computation (BRC), or a backwards pass over the node's redundant layers (its successor's layers). Unfortunately, for BRC, ~~such~~ a corresponding bubble does not exist. Eager BRC would require much extra work and data-dense communication on the critical path, which could delay training significantly (§5). As such, Bamboo runs BRC *lazily* only when a preemption actually occurs. If a node is preempted in a backward pass, continuing the pipeline requires a pause for the node's predecessor to perform BRC to restore the lost state. However, since FRC is performed eagerly, when BRC runs, much of what it needs is already in memory, keeping pauses short.

Second, performing RC increases each node's GPU memory usage. Note that the major source of the memory overhead is storing intermediate results (activations and optimizer state) from FRC, *not* the redundant layers, which take only little extra memory. To mitigate the memory issue, we leverage Bamboo's unique way of performing RC described above. Note that the purpose of saving intermediate results of a forward pass is that these results are used by its backward computation. However, in Bamboo, BRC is performed lazily upon preemptions and the intermediate results of FRC are thus not needed in normal backward passes. Hence, Bamboo swaps out the intermediate results of each node's FRC into the node's CPU memory, leading to substantial reduction in GPU memory usage. These results are swapped back into GPU memory for BRC only upon preemptions.

Bamboo continues normal training with the help of RC in the presence of non-consecutive preemptions, *i.e.*, preempted instances are not neighbors in the same data-parallel pipeline. Once consecutive instances are preempted, RC can no longer provide resilience. More redundancies could be added to provide stronger resilience, but this would incur (compute and communication) overheads that are too significant to hide. Instead, based on our empirical observation that most concurrent preemptions come from the same allocation group (*e.g.*, a zone), Bamboo takes care to ensure that consecutive nodes in each pipeline come from different zones, minimizing the chance of consecutive preemptions at a small ($<$5%) overhead (see §6.5).

Results. We built Bamboo atop DeepSpeed [51] and evaluated it by training 6 representative DNN models using EC2 spot clusters comprised of p3 instances. Compared to a baseline using on-demand instances, Bamboo delivers a **3.6**$\times$ cost reduction. Bamboo also outperforms a checkpointing approach by **3.7**$\times$. We developed a simulation

[1] In this paper, "instance" and "node" both refer to a spot instance.



(a) Pipeline parallelism (b) GPipe scheduling (c) PipeDream scheduling

Figure 1: Illustration of pipeline parallelism on a 4-node cluster: (a) the model is divided into 4 shards, each with 2 layers; (b) and (c) show the scheduling of two recent systems GPipe [24] and PipeDream [38].

framework that takes preemption traces from real spot clusters and training parameters to simulate how training progresses with larger numbers of nodes. A deep-dive with BERT across a wide range of preemption probabilities shows that the *value* (*i.e.*, performance-per-dollar) Bamboo provides stays constant and is much higher (**2.48**×) than that of on-demand instances. Bamboo is publicly available at https://github.com/uclasystem/bamboo.

2 Background

This section discusses necessary background for parallelism strategies. *Data parallelism* keeps a replica of an entire DNN on each device, which processes a subset of training samples and iteratively synchronizes model parameters with other devices. This strategy is often used with models that can fit entirely within a single GPU and used to both increase throughput or expand to batch sizes that cannot fit within a single GPU. Data parallelism can be combined with pipeline and/or model parallelism to train large models that do not fit on a single device. *Model parallelism* [13] partitions model operators across training devices. For example, the weights for a single matrix multiplication may reside across two separate GPUs, each performing a part of the full computation and then combining the results. This technique allows the model to expand beyond a single GPU by reducing the memory requirements of each operator. However, efficient model parallelism algorithms are extremely hard to design, requiring difficult choices among scaling capacity, flexibility, and training efficiency. As such, model-parallel algorithms are often architecture- and task-specific.

Pipeline parallelism [38, 24, 71] has gained much traction recently due to its flexibility and applicability to a variety of neural networks. Pipeline parallelism divides a model at the granularity of layers and assigns a shard of layers to each device. Figure 1(a) shows an example where the model is partitioned into four shards and each worker hosts one shard (with two layers). Each worker defines a computation stage and the number of stages is referred to as the *pipeline depth* (*e.g.*, 4 in the example). One worker only communicates with nodes holding its previous stage or next stage. Each input batch is further divided into *microbatches*. In each iteration, each microbatch goes through all stages in a forward pass and then returns in an opposite direction in a backward pass. There are often multiple microbatches residing in the pipeline and different nodes can process different microbatches in parallel to improve utilization.

A key challenge in efficient pipeline parallelism is how to schedule microbatches. GPipe [24] schedules forward passes of all microbatches before any backward pass, as shown in Figure 1(b) where each node processes four microbatches. This approach leaves a "bubble" (*i.e.*, white cells) in the middle of the pipeline, leading to inefficient use of compute devices. PipeDream [38] proposes the one-forward-one-backward (1F1B) schedule to interleave the backward and forward passes, as shown in Figure 1(c). 1F1B can reduce the bubble size and the peak memory usage.

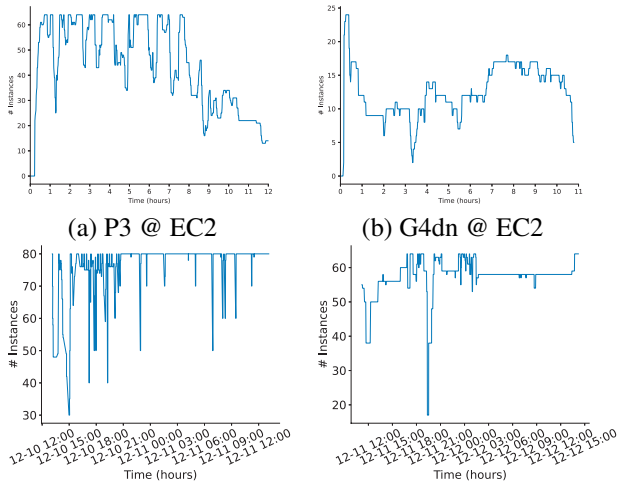
However, even with carefully-designed schedules, the pipeline bubble is still hard to eliminate. A fundamental reason is that it is extremely difficult to find the optimal layer partitioning to have each stage processed at the same rate. There exists a body of algorithms proposed recently to optimize layer partitioning and most of them are model- and hardware-specific [38, 16]. These algorithms are often time-consuming for large models, unsuitable for preemptible instances where the number of nodes keeps changing [2].

PipeDream [38] proposes asynchronous pipelining to eliminate the bubble—a node is allowed to work with stale weights to reduce the wait time. However, asynchronous microbatching introduces uncertainty in model convergence. In general, the effectiveness of synchronous v.s. asynchronous training is still open to debate. Furthermore, asynchronous training introduces inconsistencies in model state, which can create a more significant convergence issue when training occurs on preemptible instances, due to the need of frequent reconfigurations. For example, under synchronous microbatching, a reconfiguration can be performed at the end of each optimizer step (*i.e.*, parameter update), and hence the reconfigured pipelines can start with the up-to-date parameters. This is impossible to do under asynchronous microbatching.

As a result, we built Bamboo atop synchronous microbatching where model state is always consistent. Instead of attempting to reduce the bubble, we explore an orthogonal direction—how to leverage the bubble to run RC efficiently.

3 Motivation

This section motivates Bamboo from two aspects: (1) high preemption rates and unpredictability of spot instances, and (2) high performance overheads of strawman approaches.



(a) P3 @ EC2 (b) G4dn @ EC2

(c) n1-standard-8 @ GCP (d) a2-highgpu-1g @ GCP

Figure 2: Preemptions traces for a target cluster of size 64 instances on EC2 and 80 instances on GCP. Each graph shows a full-day trace for a GPU family in a cloud.

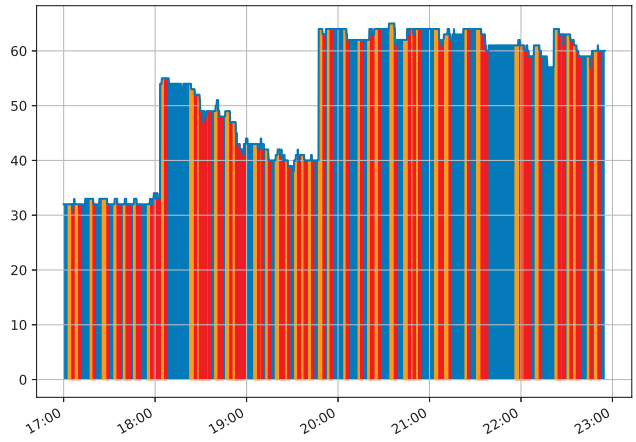


Figure 3: Training GPT-2 using checkpointing/restart with an autoscaling group of 64 P3 spot instances. Each color represents time spent in a distinct state,including Blue: training actively made progress; Orange: the cluster made progress that was then wasted; and Red: cluster restarting.

Preemptions of Spot Instances. We first studied failure models with spot instances on major public clouds. Figure 2 shows a set of real preemption traces collected from running spot instances in two public clouds: Amazon EC2 and Google Cloud Platform (GCP). For EC2, we used two GPU families: P3 (NVIDIA V100 GPUs with 32GB of memory) and G4dn (NVIDIA T4 GPUs with 16GB of memory). For GCP, we used `n1-standard-8` (NVIDIA V100 GPUs with 16GB GRAM) and `a2-highgpu-1g` (NVIDIA A100 GPUs with 40GB GRAM). For each family, we collected traces for a 24-hour window. In each experiment, we used an autoscaling group to maintain a cluster of 64 with an exception of `us-east1-c` in GCP, whose cluster size is 80. The autoscaling group, provided by each cloud, automatically allocates new instances upon preemptions to maintain the size (though without any guarantee).

From both families, node preemptions and additions are frequent and bulky (*i.e.*, many nodes get preempted at each time). This can make a checkpointing-based approach restart many times in a short window of time, leading to large inefficiencies (discussed shortly). Furthermore, both preemptions and allocations are unpredictable. While the autoscaling group attempts to allocate new nodes to maintain the user-specified size, allocations are committed incrementally; new allocations are mixed with preemptions of existing instances, making the spot cluster an extremely dynamic environment.

To understand the nature of the nodes that are preempted at the same time, we carefully analyzed two 24-hour preemption traces collected respectively from EC2 and GCP. For the EC2 trace, preemptions occur at 127 distinct timestamps, each of which see many preempted nodes. Of these 127 timestamps, only 7 see preemptions from multiple zones; at each of the remaining 120 timestamps, all nodes preempted come from the same zone. A similar observation was made on the GCP trace (12 out of 328 timestamps see cross-zone preemptions). These results confirmed the observations made by existing works [21, 20]: preemptions tend to be independent based on each individual spot market and each availability zone has a different and independent spot market—this is because each availability zone maintains capacity separately and therefore capacity preemptions in one zone are not associated with capacity preemptions in another.

These observations motivate our design—even with 1-node redundancies, Bamboo can recover from a majority of preemptions if consecutive nodes are not preempted at the same time; we maximize this possibility with a best-effort approach that makes consecutive nodes in each pipeline come from different zones. Although this may increase communication costs, it does not lead to visible performance impacts for Bamboo because Bamboo only sends (small amounts of) activations data between nodes.

Strawman #1: Checkpointing. We next show why a technique based on checkpointing and restarting does not work. We developed a new checkpointing system on top of DeepSpeed [51], providing checkpointing and restarting functionalities similar to TorchElastic [47] and Varuna [2]. We modified DeepSpeed to checkpoint *continuously* and *asynchronously*. In particular, each worker moves a copy of any relevant model state to CPU memory whenever the state is generated; the CPU then asynchronously writes it to remote storage so that training and checkpointing can fully overlap. During restarting, our system automatically adapts the prior checkpoints to the new pipeline configurations.

To understand how well this technique performs, we used it to train GPT-2 over 64 p3.2xlarge GPU spot instances on EC2. We profiled the training process and collected the check-

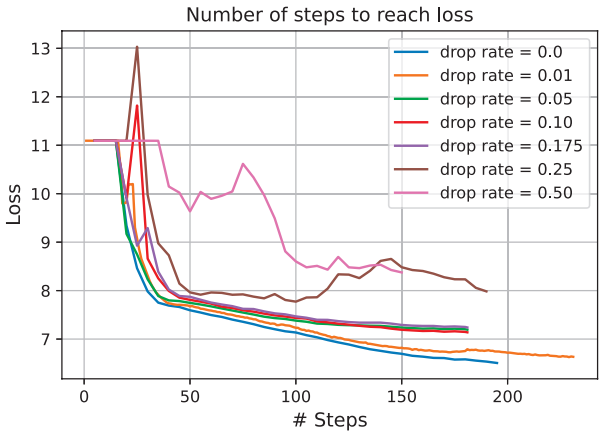


Figure 4: Effects of sample dropping under different rates.

pointing times, reconfiguration overheads, and total execution time. Figure 3 reports these results. The blue sections represent the times the system spent making actual progress for training. The red sections represent the times on reconfiguring (*i.e.*, restarting) while the orange sections show the times for wasted work—the computation that was done but not saved in checkpoints; the system ended up redoing these computations after restarting. This is because preemptions often occur during checkpointing, and hence, the system must roll back to a previous checkpoint. Frequent rollbacks slows down the training significantly. Note that systems such as Varuna and TorchElastic share this property and would have similar training patterns when facing regular preemptions. As shown, although checkpointing itself can be done efficiently, the restarting overheads (*i.e.*, for adapting existing checkpoints to new pipeline configurations) and the wasted computations take **77%** of the training time.

Strawman #2: Sample Dropping. An alternative approach that has shown promise is to take advantage of the statistical robustness of DNN training and allow some samples to be dropped so that training can continue without significant loss of accuracy [67, 36]. These techniques are also known as *elastic batching* because dropping samples is equivalent to changing the effective batch size at a training iteration (with the learning rate dynamically adjusted).

In the case of pipeline parallelism, we implemented sample dropping by suspending a pipeline upon losing an instance while letting other data-parallel pipelines continue to run. The system performs optimizer steps with the gradients of whichever data-parallel pipelines are able to complete that training step. Learning rate was adapted linearly with respect to the effective batch size to make sure that the only effect on the accuracy is the lost samples, but *not* a mismatch between hyperparameters and training configurations. In doing so, the training can continue for sometime without a reconfiguration (which is needed upon allocations).

We conducted a set of experiments to simulate the effect of sample dropping on model accuracy with a range of drop rates. Note that we could not obtain these results with the actual spot instances because we could not control the preemption rate. We ran a pre-training benchmark with GPT-2 using 16 on-demand instances from the same EC2 family, which form four data-parallel pipelines, each with four stages. To consider a range of different failure models, we used different rates of preemption to generate preemption events. Upon a preemption event, we randomly selected a pipeline and zero out the pipeline's gradients in that iteration. We measured the model's evaluation accuracy every 5 training steps. These results are shown in Figure 4 where each curve represents the function of the number of steps needed to reach a given loss for a particular drop rate.

Similarly to checkpointing, sample dropping works well for low preemption rates, but when frequent preemptions occur, many samples can be lost quickly and its impact on model accuracy quickly grows to be too significant to overlook. While this experiment was not an exact recreation of a sample dropping scenario, these results represent an *under-approximation* of the effect of the actual sample dropping (which can lose more accuracy than reported by Figure 4). This is because the actual sample dropping rate should be higher than the instance preemption rate—a preempted instance would likely be down for some time and consecutive samples would be dropped in a real setting. Note that training samples are shuffled before loading; hence, the effects of randomly dropping consecutive samples (*i.e.*, the actual scenario) and dropping random samples sporadically (*i.e.*, our experiment) should be similar.

4 Overview

Goal and Non-Goal. Our goal is *not* to automatically determine the cheapest way to train a given model (*e.g.*, which parallelism model can lead to the largest cost savings). Instead, Bamboo aims to enable efficient and preemption-safe training over cheap spot instances.

User Interface. To use Bamboo, a user specifies two system parameters D and P, as they normal would to use other pipeline-parallel systems, where D is the number of data-parallel pipelines and P is the pipeline depth. Due to the need of storing redundant layers, Bamboo requires a larger pipeline depth P than a normal pipeline-parallel system such as PipeDream [38]. We observed, empirically, that to avoid swapping data between CPU and GPU memory on the critical path, Bamboo's pipeline should be $\sim$1.5$\times$ (see §6.4) longer than an on-demand pipeline due to the extra memory needed to (1) hold the redundant layers and (2) accommodate potential pipeline adjustments. Given that spot instances are much cheaper (*e.g.*, 3-4$\times$ on EC2) than on-demand instances, training with 1.5$\times$ more nodes still leads to significantly reduced costs. While we recommend 1.5$\times$ more nodes, the number of active instances in a cluster is often much smaller due to preemptions and incremental allocations.

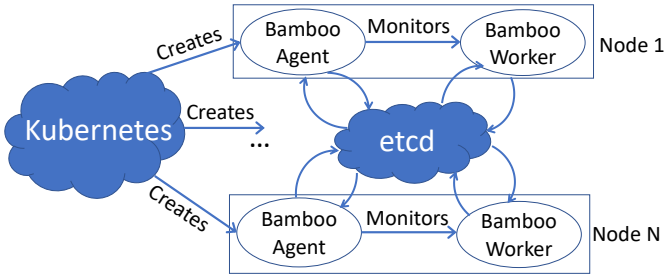


Figure 5: Bamboo runs one agent process per node (*i.e.*, spot instance). An agent monitors worker processes (each running a training script) that use our modified DeepSpeed. All workers and agents coordinate through `etcd` [42].

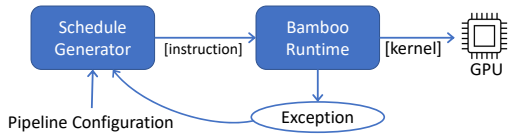


Figure 6: Bamboo worker.

$P \times D$ will be the size of the spot cluster Bamboo attempts to maintain throughout training. Preemptions can cause Bamboo to reduce the pipeline depth and/or the number of pipelines; in such cases, Bamboo would request more instances to bring the size of the cluster back to $P \times D$. However, Bamboo would never try to scale the training beyond $P \times D$. In other words, P and D are the *upper bound* of the pipeline depth and number of pipelines. It is important to note that the goal of the autoscaling framework we build for Bamboo is to adjust the pipelines *passively* in response to node preemptions and additions that we cannot control, rather than *proactively* finding an optimal cluster configuration to achieve better performance. This distinguishes Bamboo from existing works on autoscaling distributed training [43, 2, 25], whose goal was to find better configurations.

System Overview. Figure 5 shows an overview of our system. We built Bamboo on TorchElastic [47] and DeepSpeed [51]. In particular, we built the Bamboo agent, which runs on each node to kill/add a data-parallel pipeline, on top of TorchElastic. The agent monitors a Bamboo worker process on the same node, which is a DeepSpeed application enhanced with our support for redundant computation. Bamboo workers run D data-parallel pipelines that use an `all-reduce` phase to synchronize weights at the end of each iteration. Our spot instances are managed by Kubernetes [33], which is configured to automatically scale by launching a Bamboo agent on each new allocation. Our agents communicate and store cluster state on `etcd` [42], a distributed key-value store.

Each Bamboo worker uses a runtime to interpret the schedule, which produces a sequence of instructions, as shown in Figure 6. The schedule is generated statically based on the stage ID of the current worker and pipeline configurations, including the depth of pipeline and total number of micro-batches. The instructions consist of a computation component (*i.e.*, forward, backward, and apply gradient), and a communication component (*i.e.*, send/receive activation, send/receive gradient, and all-reduce). The Bamboo runtime interprets these instructions by launching their corresponding kernels on GPU. Communication instructions can fail due to preemptions. Upon a failure, the runtime throws an exception and falls back to use a failover schedule.

5 Redundant Computation

For ease of presentation, our discussion focuses on one node running one stage in the pipeline. Support for multi-GPU nodes will be discussed shortly.

Preemption of a node is detected by its neighboring nodes in the same pipeline during the execution of communication instructions. If a node on one side of the communication is preempted, the node on another side will catch an IO exception due to broken socket and update cluster state on `etcd`. Bamboo detects preemptions based on socket timeout. Although we could let a node to be preempted actively notify its neighbors in the grace period before the preemption, the length of this period varies across different clouds and hence Bamboo does not use it currently.

Since the victim node communicates with two nodes in the pipeline, both of its neighbors can catch the exception. The observed exception will be shared between these two nodes through `etcd`. This two-side detection is necessary for Bamboo to understand which node fails and generate the failover schedule. In addition to the two neighbors, nodes in other pipelines involved in the `all-reduce` operation also need to be informed. To safely perform `all-reduce`, each node participating in `all-reduce` reads the up-to-date cluster state on `etcd` and, if another pipeline has a failure, waits until the failure is handled.

5.1 Redundant Layers and Computation

To quickly recover from preemptions, Bamboo replicates the model partition on each worker node in each data-parallel pipeline. Instead of saving these replicas to a centralized remote storage (like checkpointing), Bamboo takes a *decentralized* approach by letting each node replicate its own model partition (*i.e.*, layer shard) on its predecessor node in the same pipeline. The first node has its layer replica stored on the last node in the pipeline. Conceptually, the last node is considered the "predecessor" of the first node. For simplicity of presentation, we use *forward stage IDs* to identify nodes, that is, a node that runs the forward stage $n+1$ is always considered as a successor of a node running the forward stage n (although in the backward pass, $n+1$ is a stage before n).

Our key idea is to let each node run normal (forward and backward) computation over its own layers and redundant (forward and backward) computation over the replica layers for its successor node. Let FRC_n^m/BRC_n^m denote the forward/backward redundant computation that is per-

formed on node m for node n, respectively. In Bamboo, $n = (m+1)\ mod\ P$ where P is the pipeline depth. Let FNC_n/BNC_n denote the forward/backward normal computation on node n. In Bamboo's pipeline, FRC^n_{n+1}/BRC^n_{n+1} is exactly the same computation as FNC_{n+1}/BNC_{n+1}, working with the same model parameters and optimizer states. To enable the last node to perform RC for the first node, we let it fetch input samples directly.

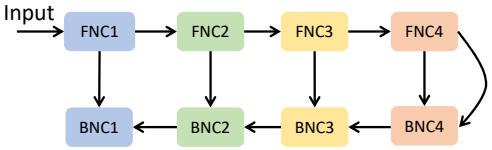


Figure 7: Dependencies between normal pipeline stages.

Why Neighboring Nodes? Due to our focus on pipeline parallelism, Bamboo performs RC on predecessor nodes to exploit *locality* for increased efficiency. To see this, we first need to understand the dependencies between different (backward and forward) pipeline stages that a microbatch goes through, as illustrated in Figure 7. For each forward stage FNC_n, it depends only on the output of its previous stage FNC_{n-1}. However, for each backward stage BNC_n, it has two dependencies: one on the output of stage BNC_{n+1} and a second on its corresponding forward stage FNC_n. The first is a *hard* dependency without which BNC_n cannot be done, while the second is a *soft* dependency primarily for efficiency—intermediate results produced by FNC_n can be reused to accelerate BNC_n. Without such cached results, BNC_n has to recompute many tensors (*i.e.*, tensor rematerialization [8]), leading to inefficiencies.

Figure 8 shows dependencies on an RC-enable pipeline where each node performs both normal and redundant (backward and forward) computation. Here solid/dashed arrows represent inter/intra-node dependencies. By running FRC for node $n+1$ on node n, *locality benefit* can be clearly seen because FRC only creates intra-node dependencies, which do not incur any extra communication overhead. However, in a backward pass, such a locality benefit does not exist for BRC^n_{n+1}, which requires the output of BNC_{n+2} and incurs much extra communication. This motivates our eager-FRC-lazy-BRC design which does not perform BRC until a preemption occurs and hence eliminates the extra communication cost in normal executions.

Note that we could also perform FRC lazily, but this would significantly increase the pause time for recovery. This is because (1) recovering from preemptions at both forward and backward pass now require a pause; and (2) lazy FRC would not produce intermediate results that can be used to speed up BRC and hence BRC's pause would be much longer. Since FRC can be scheduled in the pipeline bubble and overlap with FNC, performing it eagerly is a better choice.

The careful reader may think of an alternative approach that places node n's layer replica on node $n+1$ as opposed to

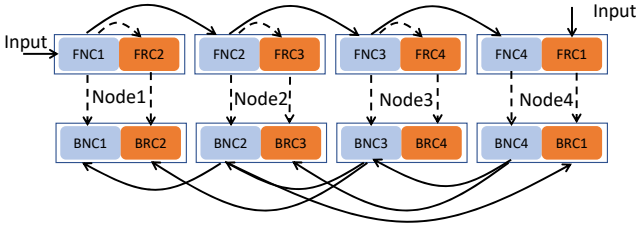


Figure 8: Dependencies between RC-enabled pipeline stages: solid/dashed arrows represent inter/intra-node dependencies; for simplicity, FRC_n/BRC_n in the figure represents FRC^{n-1}_n/BRC^{n-1}_n.

node $n-1$ (*i.e.*, its successor rather than its predecessor). This approach is symmetric to our design in that it turns inter-node dependencies for BRC into intra-node dependencies, but intra-node dependencies for FRC into inter-node dependencies. As a result, it eliminates the extra backward communication at the cost of increased forward communication. However, unlike Bamboo's design that can use lazy BRC to eliminate the extra backward communication, it is not as easy to eliminate the extra forward communication with lazy FRC—if FRC is not done eagerly in each iteration, BRC (regardless of whether it is eager or lazy) must perform tensor re-materialization, which incurs a long delay.

Level of Redundancy. As with any redundancy-based systems, the more redundancies, the higher level of resilience. For example, since Bamboo performs redundant computations only for one node, it cannot provide resilience when preemptions occur on consecutive nodes in a pipeline, in which case a reconfiguration is needed (see §A). However, enabling RC for multiple nodes can significantly increase the FRC time, making it much longer than what the bubbles can accommodate. Furthermore, the locality benefit (*i.e.*, FRC only incurs intra-node dependency) does not hold anymore, because FRC now depends on the outputs of multiple nodes. This can slow down the training substantially.

Takeaway. Storing each node's replica layers on its predecessor and running eager-FRC-lazy-BRC achieves low-overhead RC for pipeline parallelism. While this design does not support consecutive preemptions, Bamboo takes care to make consecutive nodes come from different zones. As discussed in §3, if multiple preemptions occur at the same time, the preempted nodes are highly likely to be from the same zone. As a result, our node assignment reduces the chance of consecutive preemptions, making RC effective for most preemptions. Although cross-zone data transfer can incur an overhead, this overhead is negligible (*e.g.*, <3%), as reported in Appendix §6.5, because in pipeline-parallel training, each node only passes a small amount of activation data to its neighbors.

We refer to the preempted node as the *victim node*, and the node saving the replica of the victim as its *shadow node*.

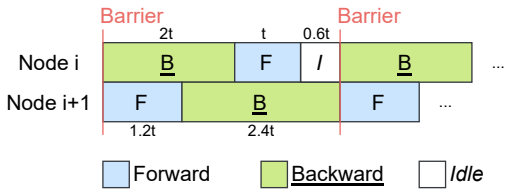


Figure 9: A closer examination of the pipeline bubble. Here we assume the forward pass on node i and $i+1$ takes time t and $1.2t$, respectively. Hence, a bubble of $0.6t$ exists before each communication barrier.

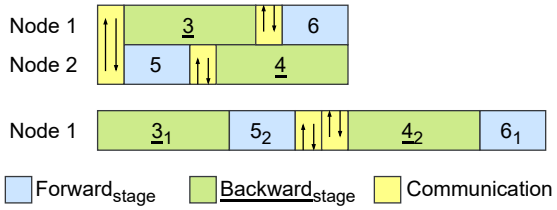


Figure 10: An example of merged instruction sequences in a failover schedule. We use PipeDream's 1F1B schedule as shown Figure 1(c), and assume node 2 is the victim node and node 1 is the shadow node.

5.2 Schedule Redundant Computation

It is straightforward to see that RC incurs an overhead in both time and memory. We propose to (1) schedule FRC into the pipeline bubble to reduce forward computation overhead, (2) perform BRC lazily to reduce backward computation/communication overhead, and (3) offload unnecessary tensors to CPU memory to reduce memory overhead.

Eager FRC. As discussed in §2, the pipeline bubble can come from either imperfect scheduling or unbalanced pipeline partitioning. To illustrate, consider Figure 9 with PipeDream's 1F1B schedule. Suppose there are two consecutive nodes in the pipeline where both the forward and the backward computation of node $i+1$ run $1.2\times$ slower than those of node i. The communication between these two nodes serves as a barrier. Since node i runs faster, it always reaches the barrier earlier and waits there until node $i+1$ arrives. This wait period is where we should schedule FRC.

Bamboo builds on the 1F1B schedule (Figure 1(a)) due to its additional efficiency compared to GPipe's schedule (Figure 1(b)). However, even for 1F1B, bubbles widely exist in a pipeline—as a microbatch passes different pipeline stages, the later a stage, the longer the (backward and forward) computation takes. This is because for the 1F1B schedule, the number of active microbatches in a later stage is always smaller than that in an earlier stage. In Figure 1 (c), for example, node 1 has 3 active microbatches while node 2 only has 2. Consequently, later stages often consume less memory. To balance memory usage, the layer partition on a later node is often larger that that on an earlier node in the pipeline, and hence a later stage runs slower. A detailed analysis of bubble size can be be found in Appendix §C.1.

Scheduling. Based on this observation, we schedule FRC on a node before the node starts communicating with its successor node. This is where a bubble exists. In cases where the FRC cannot fit entirely into the bubble (*i.e.*, for the last four stages in Figure 14), we overlap FRC and FNC as much as we can. However, for the same microbatch, FRC^n_{n+1} depends on FNC_n and they cannot run in parallel. To resolve this dependency issue, we focus on different microbatches for FNC and FRC. That is, Bamboo schedules FNC_n for the k-th microbatch and FRC^n_{n+1} for its previous $(k-1)$-th microbatch to run in parallel. Since there is no dependency between them, their executions can overlap.

To reduce memory overhead, Bamboo follows a well-known principle to offload less frequently used tensors to CPU memory. Specially, since BRC is *not* performed in normal training passes, FRC's outputs and intermediate results are not needed until a preemption occurs and BRC is triggered. As a result, we swap out these data after FRC is done for each microbatch on each node. These data take the majority of FRC's memory consumption; swapping them out significantly reduces FRC's GPU memory usage [52]. However, we leave the redundant weights in GPU memory for efficient FRC because these weights are needed for FRC on each microbatch.

Lazy BRC and Recovery. BRC is executed by a *failover schedule* which a node runs when detecting its successor node fails. In particular, for the current iteration, all the lost gradients must be re-computed, while for the following iterations, all instructions of the victim node must be executed by its shadow node (until a reconfiguration occurs). Nodes that originally communicate with the victim node are transparently rerouted to the shadow node. The failover schedule is generated by merging the schedules of the victim and shadow node. In particular, a schedule consists of a sequence of instructions and we divide it into two groups—(1) continuous communication instructions, which is placed at the head of a group and (2) computation instructions that can be executed without remote data dependencies.

When the two instruction groups (from the victim and shadow nodes) are merged, the instructions are interleaved with the following rules. (1) Communication instructions are still placed in the beginning of the merged groups. (2) Communications that used to be inter-node between the victim and the shadow are removed. (3) External communications from the victim node are first performed. (4) Computation instructions are ordered such that backward computation is always executed earlier; after the backward computation is done, the memory occupied by intermediate results is freed. Figure 10 shows an example of merged instruction sequences if node 2 is the victim node and node 1 is the shadow node.

Model	Dataset	Samples	D	P
ResNet-152 [22]	ImageNet [32]	300,000	4	12
VGG-19 [63]	ImageNet [32]	1,000,000	4	6
AlexNet [32]	ImageNet [32]	1,000,000	4	6
GNMT-16 [68]	WMT16 EN-De	200,000	4	6
BERT-Large [15]	Wikicorpus En [15]	2,500,000	4	12
GPT-2 [49]	Wikicorpus En [15]	500,000	4	12

Table 1: Our models, datasets, pipeline configurations.

Support for Multi-GPU Nodes. Bamboo's RC works for multi-GPU settings—this requires replicating all layers that belong to the GPUs of one node in the GPUs of its predecessor node. In other words, we use "group replicas" as opposed to individual replicas. However, in the presence of frequent preemptions, using multi-GPU would yield poorer performance—losing one node (with multiple GPUs) is equivalent to losing multiple nodes in the single-GPU setting. Our evaluation (§6) shows that it is much harder to allocate new multi-GPU nodes during training than single-GPU nodes.

Once Bamboo loses too many nodes or there are many idle nodes (*i.e.*, new allocations) waiting to join the pipelines, Bamboo launches a reconfiguration. Details of the reconfiguration process can be found in §A.

Support for Pure Data Parallelism. Bamboo supports pure data parallelism (without model partitioning). Due to space constraints, here we briefly discuss how it is supported. We use the same redundant computation strategy—Bamboo replicates the parameter and optimizer state of each node on a different node and uses these replicas as redundancies to provide quick recovery. For pure data parallelism, there is no bubble time to schedule RC. Eager FRC would be equivalent to overbatching (*i.e.*, each node processes its original minibatch plus a redundant minibatch). To reduce the FRC overhead and make RC fit into the GPU memory constraints, we over-provision spot instances (by 1.5×, in the same way as discussed in §5) to make each node process a smaller batch.

Enabling eager FRC doubles the batch size. However, it results only in a ∼1.5× increase in the computation time due to the parallelism provided by GPUs. This overhead can be effectively reduced by slightly over-provisioning (1.5 $\times D$) nodes, increasing the degree of parallelism and decreasing the impact of overbatching. This enables us to run FRC eagerly without incurring much overhead (*i.e.*, <10%).

Once Bamboo loses too many nodes or there are many idle nodes (*i.e.*, new allocations) waiting to join the pipelines, Bamboo launches a reconfiguration. Details of the reconfiguration process can be found in Appendix §A.

6 Evaluation

Bamboo is implemented in ∼7K LoC as a standard Python library. We evaluated Bamboo by pretraining a range of popular vision and language models, as shown in Table 1. For the first four (smaller) models that were also used in PipeDream [38] (which actually used smaller versions of these models), we took the values of D (the number of data-parallel pipelines) and P_{demand} (pipeline depth) from PipeDream [38]'s configurations.

As discussed earlier in §4, to avoid swapping Bamboo needs 1.5× more instances for each pipeline and hence each P reported in Table 1 equals $1.5\times P_{demand}$. For BERT and GPT2, we used 4 and 8×1.5=12 as D and P. We have also evaluated with another pipeline depth $P_h = P_{demand} \times \frac{Price_{demand}}{Price_{spot}}$; these results can be found in §6.2.

We trained these models on a spot cluster from EC2's p3 family where each instance has V100 GPU(s) with 16GB GPU memory and 61GB CPU memory. Each on-demand instance costs $3.06/hr per GPU while the price of its spot counter-part (at the time of our experiments) is $0.918/hr. Our evaluation uses two on-demand baselines: (1) p3 instances each with four V100 GPUs (Demand-M) and (2) p3 instances each with a single GPU (Demand-S). For both baselines, the pipeline configuration was the same and all nodes were obtained from one availability zone.

For all experiments, we trained each model to a target validation accuracy, which is a particular number of samples for the model. We did not train them to higher accuracies because large models take a huge amount of time to train (*e.g.*, weeks) to reasonable accuracies; using such a large amount of resources (even spot instances) goes beyond our financial capabilities. Furthermore, Bamboo uses synchronous training where the time per iteration is fixed; hence, training for extended time would not change our results.

For on-demand instances, we used the largest per-GPU minibatch that fits in one GPU's memory—anything larger yields out-of-memory exceptions. This ensures that we hit peak achievable FLOPs on a single device. For data-parallel runs with n workers, the global minibatch size is $n \times g$ where g is the minibatch size. The global minibatch sizes we used are consistent with those used by the ML community and reported in the literature for these models. We used a per-GPU minibatch size of 256 per GPU for VGG-19, 512 for AlexNet, 2048 for ResNet-152, 32 for GNMT-16, 256 for BERT-Large, and 256 for GPT-2. For microbatch size, we always selected a small value and tuned it for different models/configurations. We trained the vision models with an initial learning rate of 0.001, respectively, with a vanilla SGD optimizer [29]. For language models, we used the Adam optimizer [30] with an initial learning rate of $6e^{-3}$. We used half (fp16) precision in all our experiments.

6.1 Training Performance and Costs

Overall Performance. To thoroughly and deterministically evaluate Bamboo's performance over spot instances under different preemption rates, we first ran a 48-node cluster (*i.e.*, the configuration for ResNet, BERT, and GPT) and a 32-node cluster (*i.e.*, for VGG, AlexNet, and GNMT) on AWS and collected a 24-hour preemption trace for each. On these traces, the *hourly preemption rate* varies significantly, ranging from

Model	System	Time (Hours)	Throughput	Cost ($/hr)	Value
ResNet	D-M	2.78	30.00	97.92	0.31
	D-S	2.60	32.00	97.92	0.33
	B-M	[4.31, 5.31, 10.14]	[19.35, 15.69, 8.22]	[44.33, 40.01, 37.21]	[0.43, 0.39, 0.22]
	B-S	[**3.85**, 4.29, 6.87]	[**21.67**, 19.41, 12.13]	[**42.23**, 40.39, 36.72]	[**0.51**, 0.48, 0.33]
VGG	D-M	1.41	197.00	48.96	4.02
	D-S	1.66	167.00	48.96	3.41
	B-M	[2.98, 3.67, 4.33]	[93.34, 75.75, 64.22]	[21.31, 19.55, 18.43]	[4.38, 4.11, 3.48]
	B-S	[**1.81**, 2.22, 2.83]	[**153.31**, 124.88, 98.21]	[**20.19**, 19.28, 18.36]	[**7.59**, 6.48, 5.35]
AlexNet	D-M	0.77	359.00	48.96	7.33
	D-S	0.78	336.00	48.96	6.86
	B-M	[1.02, 1.34, 1.93]	[271.06, 207.43, 143.57]	[21.31, 19.55, 18.43]	[12.72, 10.61, 7.79]
	B-S	[**0.82**, 0.86, 0.99]	[**340.32**, 321.65, 280.42]	[**20.19**, 19.28, 18.36]	[**16.86**, 16.68, 15.27]
GNMT	D-M	2.06	27.00	48.96	0.55
	D-S	2.31	24.00	48.96	0.49
	B-M	[3.98, 5.13, 8.78]	[13.95, 10.82, 6.33]	[21.31, 19.55, 18.43]	[0.65, 0.55, 0.34]
	B-S	[**2.94**, 3.41, 6.31]	[**18.92**, 16.31, 8.8]	[**20.19**, 19.28, 18.36]	[**0.94**, 0.85, 0.48]
BERT	D-M	5.89	118.00	97.92	1.21
	D-S	6.43	108.00	97.92	1.10
	B-M	[9.75, 12.31, 16.66]	[71.22, 56.41, 41.68]	[44.33, 40.01, 37.21]	[1.61, 1.41, 1.12]
	B-S	[**7.02**, 8.3, 11.46]	[**98.87**, 83.70, 60.59]	[**42.23**, 40.39, 36.72]	[**2.34**, 2.07, 1.65]
GPT	D-M	4.34	32.00	97.92	0.32
	D-S	4.63	30.00	97.92	0.30
	B-M	[7.83, 9.92, 12.04]	[17.73, 14.00, 11.54]	[44.33, 40.01, 37.21]	[0.40, 0.35, 0.31]
	B-S	[**4.64**, 6.12, 10.08]	[**29.92**, 22.68, 13.78]	[**42.23**, 40.39, 36.72]	[**0.71**, 0.56, 0.38]

Table 2: Comparisons between training with DeepSpeed over on-demand instances and Bamboo over spot instances. For Bamboo, we trained each model three times, and their results are explicitly listed in the form of $[a, b, c]$ for the 10% (average), 16%, and 33% preemption rates, respectively.

no preemption all the way to 16 nodes preempted (33%), with an average rate of 4-6 nodes per hour (8-12%). To account for such changes, we extracted from each trace three segments, each with a different hourly preemption rate: 10%, 16%, and 33%. We used AWS' fleet manager to trigger preemptions by replaying these segments. Note that if we were to run Bamboo over the uncontrolled spot cluster, there would be no way to enable a direct comparison.

We trained ResNet, BERT, and GPT by replaying the three segments from the 48-node trace, and VGG, AlexNet, and GNMT by using the segements from the 32-node trace. These results are reported in Table 2. In addition to the time and monetary costs, we used a metric called *value*, which measures performance-per-dollar. Value is computed as $V = \frac{T}{C}$ where T is the training throughput, measured in terms of the number of samples per second, and C is the monetary cost per hour. Throughout the evaluation, we used both value and throughput as our metrics.

Our first observation is Demand-M slightly outperforms Demand-S due to reduced cross-node communication. However, the difference is marginal as the amount of data (*i.e.*, only activations) transferred over the network is small. Bamboo-S significantly outperforms Bamboo-M (*i.e.*, **1.4**× higher throughput and **1.5**× higher value) because (1) multi-GPU nodes are subject to more GPU failures with the same number of preemptions and (2) it is much harder to to allocate new nodes in a timely fashion.

For Bamboo-S, the results in each bracket of the form [a, b, c] show Bamboo's performance under the three preemption rates. The higher the preemption rate, the worse Bamboo's throughput and value. Given that the average preemption rate is ∼10%, the first number in each bracket (highlighted) represents Bamboo's performance on the used spot cluster. On average, Bamboo's throughput (under the 10% preemption rate) is 15% lower than DeepSpeed running over $D \times P_{demand}$ instances. There are three major reasons.

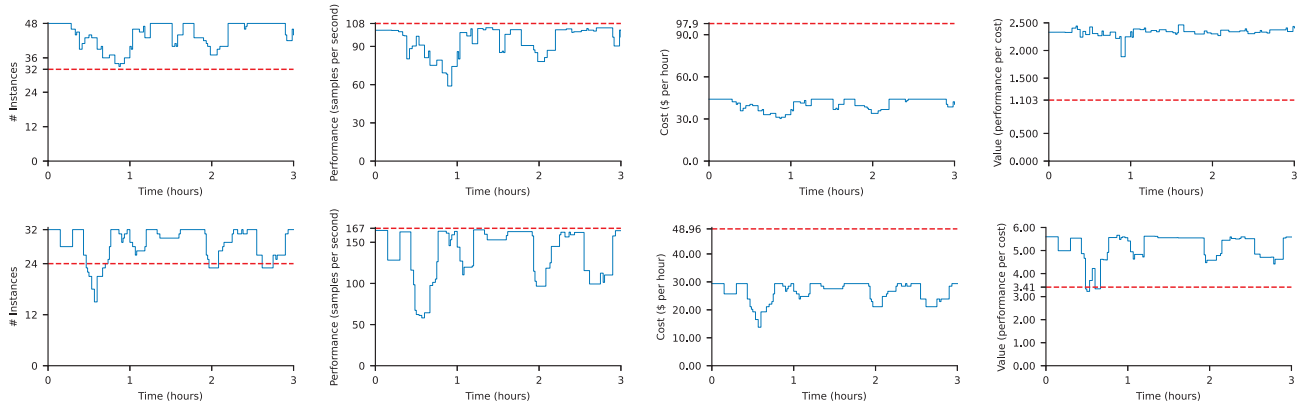
First, the number of active instances in the spot cluster is actually lower than the requested size $D \times P$. For ResNet, for example, the average number of instances throughout the training is only 25.58 although the requested cluster size is 48 (and the on-demand cluster always has 32 nodes). The autoscaling group keeps attempting to add new instances but the total number of active instances only reaches the requested size for a small period of time.

Second, Bamboo's reconfiguration contributes to reduced throughput—these overheads vary with environments and take an average of 7% of the total training time.

Third, the time for each iteration increases due to eager FRC. This is the major source of overhead for language models such as GPT-2. A detailed evaluation of RC's overhead can be found in §6.4.

Despite the small throughput reduction, Bamboo delivers an overall of **1.95**× higher value compared to training with on-demand instances. The benefit in value remains clear for five models (ResNet, VGG, AlexNet, BERT and GPUT) even when the preemption rate increases to 33% (*i.e.*, the worst-case segment of the collected trace).

To have a closer examination of Bamboo-S' training, we showed the traces for BERT-large and VGG-19, and plotted them in Figure 11. The two rows show (a) preemption traces (under the 10% rate), (b) training throughputs, (c) monetary costs, and (d) values, for BERT-large and VGG-19, respectively. Since Bamboo-M underperforms Bamboo-S, we focus on Bamboo-S in the rest of the evaluation.



(a) Trace (b) Training Throughput (c) Monetary Cost (d) Value

Figure 11: Bamboo's training performance for BERT (top) and VGG (bottom), compared to on-demand instances (red lines).

Prob.	Prmt (#)	Inter. (hr)	Life (hr)	Fatal Fail. (#)	Nodes (#)	Thruput	Cost ($/hr)	Value
0.01	8.50	2.08	15.20	0.06	45.18	87.99	41.11	2.10
0.05	48.15	0.44	10.14	0.23	43.65	76.35	39.73	1.90
0.10	99.77	0.23	6.71	0.29	41.69	72.12	37.94	1.88
0.25	276.52	0.10	3.13	1.04	35.80	60.12	32.58	1.82
0.50	709.83	0.06	1.49	5.98	26.96	40.37	24.53	1.59

(a) Results of simulating training BERT *until completion*; each preemption probability ran 1,000 times.

Prob.	Thruput	Cost ($/hr)	Value
0.01	54.87	90.73	0.60
0.05	50.66	87.43	0.58
0.10	49.18	83.23	0.59
0.25	40.59	71.24	0.57
0.50	26.24	53.05	0.49

(b) Simulation results of training BERT-large with pipeline depth P_h (which is $3.3\times P_{demand}$).

Table 3: Simulation results for more configurations.

6.2 Different Failure Models

This section demonstrates Bamboo's ability to affordably train large DNNs across a wide range of failure models. To this end, we developed an offline simulation framework that takes as input (1) the preemption probability (including preemption frequency and the number of preemptions in each bulk), (2) per-iteration training time, and (3) Bamboo's recovery and reconfiguration time, automatically calculating training performance, costs, and values. Here we focus on BERT-large and simulated its training until completion.

We experimented using 5 different preemption probabilities (*i.e.*, preemption rate per hour), and kept the preemption probability constant throughout the entire run (as opposed to replaying traces). To mimic realistic spot instance creation and preemption, we randomly generated different creation probabilities per hour and also randomly picked zones for allocations. For each preemption probability, Table 3a reports the average numbers of preemptions, intervals (*i.e.*, average time, in hours, between preemption events), average lifetime of an instance (in hours), average numbers of fatal failures (which require a restart from a checkpoint), average numbers of instances in the cluster, throughput (*i.e.*, #samples per second), costs, and values, across 1,000 simulations.

Our simulations show that Bamboo's values match our real-world runs as just reported in §6.1. Further, regardless of the preemption probability, the value of Bamboo remains stable and is constantly higher than that of training with on-demand instances (which is 1.1). This is because most preemptions can be quickly recovered without introducing much overhead. The higher the preemption probability, the less the active instances running training jobs; this is the major source of the performance slowdown. However, the cost is reduced also proportionally, leading to stable values.

Simulation for P_h. To understand the tradeoff in choosing P, we experimented with another value of P for BERT-large: P_h, which is $\frac{3.06}{0.918} \times P_{demand}$. This configuration represents the *upper-bound* of the spot training resources that can be obtained within the cost of training with P_{demand} on-demand instances (while D remains unchanged). Note that in practice the number of active instances can barely reach the requested size and hence the cost of using a spot cluster of size $P_h \times D$ is often still much lower than training with an on-demand cluster of size $P_{demand} \times D$.

To avoid incurring a large monetary cost, we used the same simulator to run this experiment. These results are reported in Table 3b. As shown, using P_h actually decreases both throughput (compared to 84 under P in Table 2) and value (due to significantly increased costs). This is because using too a large pipeline leads to poorer partitioning, underutilized resources and inferior performance.

6.3 Comparisons with Other Systems

We have reported the performance of training GPT-2 with asynchronous checkpointing and restart in Figure 3—the checkpointing-based approach spent only 23% on actual training, while Bamboo increases this percentage to **84%**. In fact, as shown in Table 3a, even for the preemption rate of 0.5, there are only 5.98 fatal failures that would require checkpoint-

ing/restart under Bamboo. On the contrary, a checkpointing-based approach would need to restart the pipeline for every one of the 709.83 preemptions. Similarly, sample dropping significantly slows down the training when the preemption rate increases, as shown in Figure 4.

Varuna. Varuna [2] is a system developed concurrently with Bamboo to enable training on spot instances. As with other existing techniques, Varuna provides resilience with checkpointing. We set up Varuna on the same spot cluster on AWS EC2 as we used in § 6.1. We ran Varuna with a $D \times P$ pipeline (*i.e.*, the same as on-demand instances) because Varuna does not use redundancies and hence not need to over-provision resources.

We trained BERT on Varuna with the same configurations, including the same datasets, model architectures, float precision, preemption rates, and hyperparameters. Varuna hung under the 33% preemption rate. For the 10% and 16% preemption rates, comparisons between Varuna and Bamboo-S are reported in Figure 12. As shown, Bamboo-S outperforms Varuna by **2.5**× and **2.7**× in throughput, respectively, under the 10% and 16% rates; and by **1.67**× and **1.64**×, in value, under the two rates. Note that value benefits are lower than throughput benefits due to Varuna's use of fewer instances. To understand the cause of Varuna's slowdown, see §3. Varuna follows a similar pattern, having to frequently restart and redo lost computations.

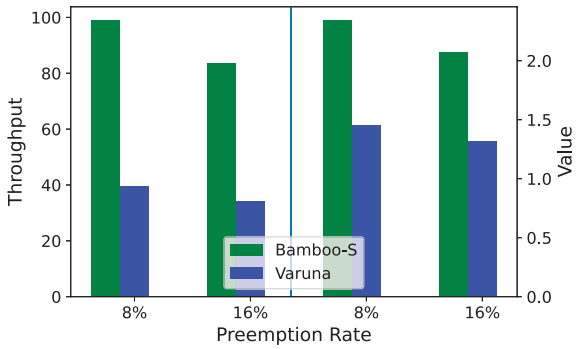


Figure 12: Throughput and value for Bamboo-S and Varuna running BERT at different preemption levels. Varuna hung at the 33% preemption rate.

6.4 Microbenchmarks of Redundant Computation

To fully understand the overhead introduced by RC, we compared time and memory among three versions of RC: eager-FRC-lazy-BRC (EFLB, Bamboo's approach), eager-FRC-eager-BRC (EFEB), and lazy-FRC-lazy-BRC (LFLB), when training BERT and ResNet. Since the focus here is the RC overhead, we ran this experiment over on-demand instances.

Table 4 reports RC's time overheads for the three RC settings. As expected, LFLB incurs the lowest per-iteration overhead because neither FRC nor BRC is performed with normal training iterations. The ∼7% overhead comes primarily from the extra code executed to prepare for a failover schedule. However, the recovery time is much longer under LFLB than the other two settings (discussed shortly). On the contrary, EFEB has the highest per-iteration overhead due to the eager execution of both FRC and BRC. The overhead incurred by EFLB, as used in Bamboo, is slightly higher than LFLB but much lower than EFEB. This is because eager FRC does not incur extra communication overhead and much its computation overhead can be hidden by scheduling it into the pipeline bubble and overlapping it with FNC.

Redundancy Mode	BERT	ResNet
Lazy-FRC-Lazy-BRC	7.01%	7.65%
Eager-FRC-Lazy-BRC (Bamboo)	19.77%	9.51%
Eager-FRC-Eager-BRC	71.51%	64.24%

Table 4: Time overhead with different RC settings.

Another interesting observation is the overhead for ResNet is lower than for BERT. This is because ResNet's layer partitioning is much more imbalanced than that of BERT (which is a transformer model where most the middle layers are equivalent). As a result, the bubble in ResNet's pipeline is much larger and hence it can accommodate a more significant fraction of FRC.

Eager FRC incurs an overall ∼1.5× overhead in GPU memory (that is why Bamboo recommends creating pipelines with 1.5× more nodes) while lazy FRC does not incur any memory overhead.

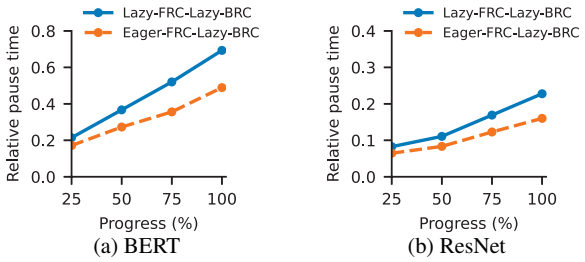


Figure 13: Relative pause time for BERT and ResNet under different RC settings. Bamboo runs into a pause when a pipeline stops training and waits for the shadow node to recover the lost state on the victim node.

To understand the pause time under these different RC settings, Figure 13 shows the relative pause time (*i.e.*, the actual pause time relative to the time of each training iteration without preemptions). As shown, lazy FRC reduces pause time by ∼35% despite the slightly higher per-iteration overhead it introduces. In summary, eager-FRC-lazy-BRC strikes the right balance between overhead and pause time.

Model	Config	Throughput	Total Transferred Bytes
BERT	Spread	148.923	16.39 GiB
BERT	Cluster	151.124	16.39 GiB
VGG19	Spread	160.12	11.213 GiB
VGG19	Cluster	165.77	11.213 GiB

Table 5: Comparison of throughput when running across availability zones compared to running within a single zone.

6.5 Cross-Zone Communication

Because Bamboo allocates workers across availability zones to minimize the probability of reconfigurations, we measured the overhead incurred by cross-zone communication. We ran Bamboo in two configurations: (1) with nodes distributed across all zones (*i.e.*, Spread) and (2) in a single availability zone with AWS' "Placement Group" option set to "Cluster" (*i.e.*, Cluster), and measured their performance differences. As reported in Table 5, the differences between these two configurations are quite low (*i.e.*, usually less than 5%). This demonstrates Bamboo's choice of assigning nodes from different availability zones as consecutive nodes in each pipeline has little impact on training performance.

7 Related Work

Parallel Training. Data parallelism [28, 14, 32, 7, 12, 72, 35, 72] is the most common parallelism model that partitions the dataset and trains on each partition. The learned weights are synchronized via either an all-reduce approach [7] or parameter servers [35, 10]. Model parallelism [14, 31, 43, 60, 62] partitions the operators in a DNN model across multiple GPU devices, with each worker evaluating and performing updates for only a subset of the model's parameters for all inputs. Recently, pipeline parallelism [24, 38, 71, 65] has been proposed to train large models by partitioning layers across workers and uses microbatches to saturate the pipeline. Popular DL training libraries such as DeepSpeed [51] and Megatron [40] support 3D parallelism, which combines data parallelism, model parallelism, and pipeline parallelism to train models at extremely large scale with improved compute and memory efficiency. Furthermore, DeepSpeed offers ZeRO-style data parallelism [52], which partitions model states across GPUs and uses communication collectives to gather individual parameters when needed.

Elastic Training. Distributed training experiences frequent resource changes. There are a number of systems [43, 21, 47, 23, 48, 25] built to provide elasticity for training over changing resources. TorchElastic [47] is a PyTorch [44]-based tool that can dynamically kill or add data-parallel workers. Huang *et al.* [23] considers elasticity for declarative ML on MapReduce, which does not work for modern deep learning workloads. Litz [48] is a system that provides elasticity in the context of CPU-based machine learning using the parameter servers. Or *at al.* [43] presents an autoscaling system built on top of TensorFlow [1] and Horovod [55], which dynamically adapts the batch size and reuses existing processes.

Exploiting Spot Instances. Proteus [21] exploits dynamic pricing on public clouds in order to lower costs for machine learning workloads through elasticity. Since Proteus does not explicitly consider modern deep learning workloads, Proteus simply reprocesses the input of a preempted node with another node. Varuna [2] is a system built concurrently with Bamboo for distributed training over spot instances. However, Varuna focuses on elasticity, not quick recovery from preemptions. Bamboo, on the contrary, is designed specifically to deal with frequent preemptions.

There exists a body of work on enabling low latency and/or SLO guarantees when using preemptible spot instances. Tributary [20] is an elastic control system that exploits preemptible resources to reduce cost with SLO guarantees. Kingfisher [59] proposes a cost-aware resource acquisition scheme that uses integer linear programming to determine a service's resource footprint among a heterogeneous set of non-preemptible instances with fixed prices. Flint [56] is a system that runs batch-based data-intensive jobs on transient servers. SpotCheck [58] selects spot markets to acquire instances in while always bidding at a configurable multiple of the spot instance's corresponding on-demand price. BOSS [70] hosts key-value stores on spot instances by exploiting price differences across pools in different data-centers. ExoSphere [57] is a virtual cluster framework for spot instances. These systems are all orthogonal to Bamboo that is built specifically for deep learning training.

GPU Scheduling. There is also a large body of work on GPU scheduling [61, 69, 74, 46, 37, 19, 39, 40, 34, 73] for ML workloads. These techniques are orthogonal to Bamboo —they all focus on efficiency and throughput while Bamboo aims to perform redundant computation at a low cost.

8 Conclusion

Bamboo is the first distributed system that uses redundant computation to provide resilience and fast recovery for training DNNs over preemptible instances. An evaluation with 6 representative models shows that Bamboo provides a much higher value than (1) training on on-demand instances and (2) training with checkpointing/restart on spot instances.

Acknowledgement

We thank the anonymous reviewers for their valuable and thorough comments. We are grateful to our shepherd Yiting Xia for her feedback. This work is supported by NSF grants CNS-1703598, CNS-1763172, CNS-1907352, CNS-2007737, CNS-2006437, CNS-2128653, CNS-2106838, CNS-2147909, CNS-2152313, CNS-2151630, CNS-2140552, and CNS-2153449, ONR grant N00014-18-1-2037, a Sloan Research Fellowship, and research grants from Cisco.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: a system for large-scale machine learning. In *OSDI*, pages 265–283, 2016.

[2] S. Athlur, N. Saran, M. Sivathanu, R. Ramjee, and N. Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *EuroSys*, 2021.

[3] AWS. Amazon ec2 spot instances pricing. https://aws.amazon.com/ec2/spot/pricing/, 2021.

[4] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin. PipeSwitch: fast pipelined context switching for deep learning applications. In *OSDI*, pages 499–514, 2020.

[5] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Lightweight preemptible functions. In *USENIX ATC*, pages 465–477, 2020.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language Models are Few-Shot Learners. In *NIPS*, 2020.

[7] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. In *ICLR Workshop Track*, 2016.

[8] T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost, 2016.

[9] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cuDNN: efficient primitives for deep learning, 2014.

[10] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, 2014.

[11] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *SIGOPS Oper. Syst. Rev.*, 53(1):14–25, 2019.

[12] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing. GeePS: scalable deep learning on distributed GPUs with a gpu-specialized parameter server. In *EuroSys*, 2016.

[13] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1223–1231, Lake Tahoe, Nevada. Curran Associates Inc., 2012.

[14] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1223–1231, 2012.

[15] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL: http://arxiv.org/abs/1810.04805.

[16] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia, L. Diao, X. Liu, and W. Lin. DAPPLE: a pipelined data parallel approach for training large models. In *PPoPP*, pages 431–445, 2021.

[17] W. Fedus, B. Zoph, and N. Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *CoRR*, 2021.

[18] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: training ImageNet in 1 hour. *CoRR*, abs/1706.02677, 2017. URL: http://arxiv.org/abs/1706.02677.

[19] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: a gpu cluster manager for distributed deep learning. In *NSDI*, pages 485–500, 2019.

[20] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons. Tributary: spot-dancing for elastic services with latency SLOs. In *USENIX ATC*, pages 1–14, 2018.

[21] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons. Proteus: agile ML elasticity through tiered reliability in dynamic resource markets. In *EuroSys*, 2017.

[22] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[23] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *SIGMOD*, pages 137–152, 2015.

[24] Y. Huang, Y. Cheng, D. Chen, H. Lee, J. Ngiam, Q. V. Le, and Z. Chen. Gpipe: efficient training of giant neural networks using pipeline parallelism. *CoRR*, abs/1811.06965, 2018. URL: http://arxiv.org/abs/1811.06965.

[25] C. Hwang, T. Kim, S. Kim, J. Shin, and K. Park. Elastic resource sharing for distributed deep learning. In *NSDI*, pages 721–739, 2021.

[26] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, and I. Stoica. Checkmate: breaking the memory wall with optimal tensor rematerialization. In *MLSys*, pages 497–511, 2020.

[27] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *USENIX ATC*, pages 947–960, 2019.

[28] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. In *MLSys*, 2019.

[29] J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Annals of Mathematical Statistics*, 23:462–466, 1952.

[30] D. P. Kingma and J. Ba. Adam: a method for stochastic optimization, 2014.

[31] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014. URL: http://arxiv.org/abs/1404.5997.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[33] Kubernetes: an open-source system for automating deployment, scaling, and management of containerized applications. https://kubernetes.io/, 2021.

[34] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: opening the black box of machine learning prediction serving systems. In *OSDI*, pages 611–626, 2018.

[35] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, 2014.

[36] H. Lin, H. Zhang, Y. Ma, T. He, Z. Zhang, S. Zha, and M. Li. Dynamic Mini-batch SGD for Elastic Distributed Training: Learning in the Limbo of Resources. *CoRR*, 2019.

[37] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: fair and efficient GPU cluster scheduling. In *NSDI*, pages 289–304, 2020.

[38] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *SOSP*, pages 1–15, 2019.

[39] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *OSDI*, pages 481–498, 2020.

[40] D. Narayanan, M. Shoeybi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro, A. Phanishayee, and M. Zaharia. Efficient large-scale language model training on gpu clusters using Megatron-LM. In *SC*, 2021.

[41] A. Newell, D. Skarlatos, J. Fan, P. Kumar, M. Khutornenko, M. Pundir, Y. Zhang, M. Zhang, Y. Liu, L. Le, B. Daugherty, A. Samudra, P. Baid, J. Kneeland, I. Kabiljo, D. Shchukin, A. Rodrigues, S. Michelson, B. Christensen, K. Veeraraghavan, and C. Tang. RAS: continuously optimized region-wide datacenter resource allocation. In *SOSP*, pages 505–520, 2021.

[42] Operating etcd clusters for Kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/, 2021.

[43] A. Or, H. Zhang, and M. Freedman. Resource elasticity in distributed deep learning. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *MLSys*, volume 2, pages 400–411, 2020.

[44] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: an imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, 2019.

[45] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, pages 109–116, 1988.

[46] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *EuroSys*, 2018.

[47] PyTorch Developers. TorchElastic. 2021. URL: https://pytorch.org/docs/stable/distributed.elastic.html.

[48] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing. Litz: elastic framework for High-Performance distributed machine learning. In *USENIX ATC 18*, pages 631–644, 2018.

[49] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. In 2019.

[50] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language Models are Unsupervised Multitask Learners, 2019.

[51] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He. ZeRO: memory optimizations toward training trillion parameter models. In *SC*, 2020.

[52] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. 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.

[53] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. On parallelizability of stochastic gradient descent for speech dnns. In *ICASSP*, pages 235–239, 2014.

[54] F. Seide, H. Fu, J. Droppo, G. Li, and D. Yu. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*, Sept. 2014.

[55] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018. URL: http://arxiv.org/abs/1802.05799.

[56] P. Sharma, T. Guo, X. He, D. E. Irwin, and P. J. Shenoy. Flint: Batch-Interactive Data-Intensive Processing on Transient Servers. In *EuroSys*, 2016.

[57] P. Sharma, D. Irwin, and P. Shenoy. Portfolio-driven resource management for transient cloud servers. In *SIGMETRICS*, page 59, 2017.

[58] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. SpotCheck: designing a derivative IaaS cloud on the spot market. In *EuroSys*, 2015.

[59] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In *ICDCS*, pages 559–570, 2011.

[60] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. A. Hechtman. Mesh-TensorFlow: Deep Learning for Supercomputers. In *NIPS*, 2018.

[61] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *SOSP*, pages 322–337, 2019.

[62] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR*, 2019.

[63] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In Y. Bengio and Y. LeCun, editors, *ICLR*, 2015.

[64] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, 2005.

[65] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu. Dorylus: affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *OSDI*, pages 495–514, 2021.

[66] I. Turc, M. Chang, K. Lee, and K. Toutanova. Well-Read Students Learn Better: The Impact of Student Initialization on Knowledge Distillation. *CoRR*, 2019.

[67] T. Wang, J. Huan, and B. Li. Data dropout: optimizing training data for convolutional neural networks. In *ICTAI*, pages 39–46, 2018.

[68] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean. Google's neural machine translation system: bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL: http://arxiv.org/abs/1609.08144.

[69] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. AntMan: dynamic scaling on GPU clusters for deep learning. In *OSDI*, pages 533–548, 2020.

[70] Z. Xu, C. Stewart, N. Deng, and X. Wang. Blending on-demand and spot instances to lower costs for in-memory storage. In *INFOCOM*, pages 1–9, 2016.

[71] B. Yang, J. Zhang, J. Li, C. Ré, C. R. Aberger, and C. D. Sa. PipeMare: Asynchronous Pipeline Parallel DNN Training. In *MLSys*, 2019.

[72] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing. Poseidon: an efficient communication architecture for distributed deep learning on GPU clusters. In *USENIX ATC*, pages 181–193, 2017.

[73] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. SLAQ: quality-driven scheduling for distributed machine learning. In *SoCC*, pages 390–404, 2017.

[74] Q. Zhang, R. Zhou, C. Wu, L. Jiao, and Z. Li. Online scheduling of heterogeneous distributed machine learning jobs. In *MobiHoc*, pages 111–120, 2020.

A Pipeline Reconfiguration

Reconfiguration introduces a much longer pause to the training process than recovering using RC. The goal of reconfiguration is to rebalance pipelines so they can withstand more failures as training progresses and continue to yield good performance. Reconfiguration also attempts to allocate more instances to maintain the cluster size. As shown in §3, asynchronous checkpointing is very efficient (but frequent restarting is not), and hence, Bamboo periodically checkpoints the model state. These checkpoints will not be used unless Bamboo restarts the training from a rare fatal failure (*i.e.*, too many nodes are preempted so that training cannot continue).

Reconfiguration Triggering. Reconfiguration is triggered immediately when (1) consecutive preemptions occur simultaneously and (2) Bamboo determines that there is an urgent need to rebalance the pipelines at the end of an optimizer step. To do (2), the workers retrieve the cluster state from `etcd`, allowing them to see how many preemptions have occurred and in which pipeline they have occurred. They can also see how many workers are currently waiting to join the next rendezvous.

There are two main conditions for triggering reconfiguration at the end of an optimizer step: (a) the cluster has gained enough new nodes to reconstruct a new pipeline, and (b) Bamboo has encountered many preemptions and is close to a critical failure in the next step (*e.g.*, encountering another preemption would cause us to suspend training), in which case we must pause the training to allocate more nodes.

Reconfiguration Policy. Bamboo attempts to maintain the pipeline depth P specified by the user. Therefore, our top priority at a reconfiguration is to reestablish a full pipeline of depth P. In this case, if we have had F failures and J ($> F$) nodes are waiting to join the cluster (*i.e.*, new allocations arrive as Bamboo runs on the "spare tire"), we can fully recover all pipelines to depth P. The remaining $(J-F)$ nodes are placed in a standby queue to provide quick replacement upon future failures. However, if the number of nodes joining is smaller than F, we may end up having a number of N nodes such that $N\%P \neq 0$. In this case, instead of creating asymmetric pipelines (which complicates many operations), we move some nodes into the standby queue and decrease the total number of data-parallel pipelines. A final case is that the number of nodes joining, together with those in the standby queue, can form a new pipeline, and in this case we add a new pipeline to the system. In all these cases, the redundant layers are redistributed among the set of nodes participating in the updated pipelines.

How to Reconfigure. Once a reconfiguration is triggered, each node must be assigned a new stage (with new layers, state, and redundancies); it also needs to figure out if it will need to send or receive model and optimizer state from other nodes. Whichever nodes hits the rendezvous barrier first decides the new cluster configuration and puts the decision on `etcd` for all other nodes to read. To minimize the amount of data sent in layer transfer, Bamboo transfers layers in such a way that each node can reuse its old model and optimizer state as much as possible.

B Support for Pure Data Parallelism

Bamboo supports pure data parallelism (without model partitioning). Due to space constraints, here we briefly discuss how it is supported. We use the same redundant computation strategy—Bamboo replicates the parameter and optimizer state of each node on a different node and uses these replicas as redundancies to provide quick recovery. For pure data parallelism, there is no bubble time to schedule RC. Eager FRC would be equivalent to overbatching (*i.e.*, each node processes its original minibatch plus a redundant minibatch). To reduce the FRC overhead and make RC fit into the GPU memory constraints, we over-provision spot instances (by 1.5×, in the same way as discussed in §5) to make each node process a smaller batch.

Enabling eager FRC doubles the batch size. However, it results only in a ∼1.5× increase in the computation time due to the parallelism provided by GPUs. This overhead can be effectively reduced by slightly over-provisioning ($1.5\times D$) nodes, increasing the degree of parallelism and decreasing the impact of overbatching. This enables us to run FRC eagerly without incurring much overhead (*i.e.*, <10%).

C Additional Experiments

C.1 Bubble Size

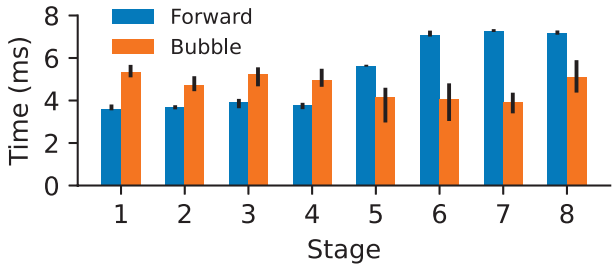


Figure 14: Comparison between bubble sizes and forward computations.

We measured the sizes of the pipeline bubble and forward computation of BERT with the same configuration as mentioned in Section 6, running on on-demand instances each with a single GPU. We manually inserted a barrier before each peer-to-peer communication, treating the time spent on the corresponding NCCL kernel as the bubble size. These results are reported in Figure 14.

To make memory evenly distributed across stages, more layers are placed on the last few stages. This explains the growth of forward computation. In this pipeline, for the first 4 stages, the bubble time is long enough to fit the entire FRC

(*i.e.*, the bubble at stage 1 should run the forward computation for stage 2). For the last 4 stages, the bubble time is shorter than the forward computation time—it can still cover ∼60% of its FRC. The rest of the FRC on these nodes is run in parallel with their regular forward computation, as discussed in §5.2.

C.2 Bamboo for Pure Data Parallelism

We ran two relatively small models such as VGG and ResNet using pure data parallelism with 8 workers (*i.e.*, we partition the data but not the model). For Bamboo, we similarly over-provisioned 1.5× additional workers. We implemented another baseline *Checkpoint*, which periodically checkpoints model state for each worker and restarts the worker on another node when its original node is preemption. We used the same global batch size for these models as reported in §6. The comparisons between Bamboo, *Checkpoint*, and on-demand training are shown in Table 6.

Note that our implementation of *Checkpoint* assumes that there is always a standby node that is ready to join and load the checkpoint (which is a unrealistic over-approximation of the allocation model on any spot market); as such, the training cost remains unchanged and its throughput is reduced as the preemption rate increases.

Model	System	Throughput	Cost ($/hr)	Value
ResNet	Demand	24.51	24.48	1.01
	Checkpoint	[12.26, 8.42, 5.03]	[7.34, 7.34, 7.34]	[1.67, 1.15, 0.68]
	Bamboo	[**21.22**, 18.31, 12.31]	[**10.56**, 10.09, 9.18]	[**2.01**, 1.84, 1.34]
VGG	Demand	144.28	24.48	5.89
	Checkpoint	[83.21, 67.21, 45.31]	[7.34, 7.34, 7.34]	[11.33, 9.15, 6.17]
	Bamboo	[**125.59**, 96.51, 73.73]	[**10.56**, 10.09, 9.18]	[**11.89**, 9.56, 8.03]

Table 6: Comparison between pure data-parallel training over on-demand instances, a checkpoint-based approach on spot instances, Bamboo on spot instances. For *Checkpoint* and Bamboo, we trained each model three times, and their results are explicitly listed in the form of [a, b, c] for the 10% (average), 16%, and 33% preemption rates, respectively.

As shown, Bamboo outperforms *Checkpoint* by **1.64**× and **1.22**× in throughput and value. Both *Checkpoint* and Bamboo deliver a higher value than on-demand training (by **2**× and **1.79**×).

We make two observations on these numbers. First, Bamboo incurs a higher cost than *Checkpoint* due to resource over-provisioning. However, as discussed above, *Checkpoint* assumes the availability of standby nodes. In practice, guaranteeing such availability requires over-provisioning as well, but we did not take this into account when calculating costs (because it is hard to know exactly how many nodes we should over-provision). Hence, the cost and value reported for *Checkpoint* are the *lowerbound* and *upperbound* of those that can be achieved by any practical implementation of a checkpoint-based approach.

Second, *Checkpoint* works much better for pure data parallelism than for pipeline parallelism (as discussed in §3). This is because recovering from a checkpoint in pure data-parallel training is much easier than pipeline-parallel training where a pipeline reconfiguration process is needed for each restart.