Checkpoint-Free Fault Tolerance for Recommendation System Training via Erasure Coding

Kaige Liu

CMU-CS-20-140

Dec 2020

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Rashmi K. Vinayak, Chair
Phillip Gibbons

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

Copyright © 2020 Kaige Liu

Keywords: Recommendation systems, erasure coding, machine learning, fault tolerance

Abstract

Deep-learning-based recommendation models (DLRMs) are widely deployed to serve personalized content to users. DLRMs are large in size due to their use of embedding tables, and are trained by distributing the model across the memory of tens or hundreds of servers. Checkpointing is the predominant approach used for fault tolerance in these systems. However, it incurs significant training-time overhead both during normal operation and when recovering from failures. As these overheads increase with DLRM size, checkpointing is slated to become an even larger overhead for future DLRMs.

In this thesis, we present ECRM, a DLRM training system that achieves efficient fault tolerance using erasure coding. ECRM chooses which DLRM parameters to encode and where to place them in a training cluster, correctly and efficiently updates parities during normal operation, and recovers from failure without pausing training and while maintaining consistency of the recovered parameters. The design of ECRM enables training to proceed without any pauses both during normal operation and during recovery. We implement ECRM atop XDL, an open-source, industrial-scale DLRM training system. Compared to checkpointing, ECRM reduces training-time overhead by up to 88%, recovers from failures significantly faster, and allows training to proceed during recovery. These results show the promise of erasure coding in imparting efficient fault tolerance to training current and future DLRMs.

Acknowledgments

I would like to thank my advisor, Rashmi Vinayak, for providing guidance for the direction of my research and patiently resolving my concerns. I would like to thank my mentor Jack Kosaian, for giving me invaluable suggestions, providing essential feedback and resolving my concerns. I would like to thank Phillips Gibbons, the instructor for Advanced Distributed & Operating Systems course, during which I had performed an early exploration of this direction and used it as my course project , for providing thorough feedback and critiques. I would like to thank my course project partner Anlun Xu for his fundamental contribution to the early stage of this work.

Contents

1 Introduction **1**

2 Background and Motivation **5**
2.1 DLRM training systems 5
2.2 Checkpointing and its downsides 6
2.2.1 Time penalty during normal operation 6
2.2.2 Time penalty during recovery 8
2.3 Fault tolerance via proactive redundancy? 8
2.3.1 Replication 8
2.3.2 Erasure codes: proactive, low-overhead 9

3 ECRM: erasure-coded training **11**
3.1 Overview of ECRM 11
3.2 Encoding and placing parity parameters 12
3.3 Correctly and efficiently updating parities 14
3.3.1 Challenges in keeping up-to-date parities 14
3.3.2 Difference propagation 15
3.4 Pause-free recovery from failure 16
3.4.1 Challenges in erasure-coded recovery 17
3.4.2 Training during recovery in ECRM 17
3.5 Maintaining consistency of recovered DLRM 18
3.6 Tradeoffs in ECRM 19

4 Evaluation **21**
4.1 Evaluation setup 21
4.2 Performance during recovery 23
4.3 Performance during normal operation 25

5 Related Work **31**
5.1 DLRM training and inference systems 31
5.2 Checkpointing 31
5.3 Coding in machine learning systems 32

6 Conclusion **33**

Bibliography 35

List of Figures

1.1 Example of the distributed setup used to train DLRMs. 2
1.2 Naive erasure-coded DLRM with $k = 3$ and $r = 1$. 2

2.1 Time required to read and write checkpoints 7
2.2 Effect of checkpointing on total training time 7
2.3 Example of ECRM with $k = 3$, $r = 1$. 9

3.1 Components and operation of a server in ECRM. Shaded boxes store data, and unshaded boxes are used for control flow. 12

4.1 Throughput when recovering from failure at 10 minutes. 22
4.2 Training progress (bottom) when recovering from failure at 10 minutes. 23
4.3 Time to fully recover a failed server. 24
4.4 Effects of the number of partitions on recovery time 25
4.5 Training-time overhead in the absence of failures 26
4.6 Throughput of training Criteo-2S-2D 27
4.7 Progress of training Criteo-2S-2D 28
4.8 Average training throughput with varying number of workers during normal operation 29

List of Tables

1.1 Alibaba's DLRM sizes. 2

3.1 Example timeline that results in ECRM inconsistency. 19

Chapter 1

Introduction

Recommendation systems are currently deployed for a variety of tasks at large internet companies. In general, a recommendation system seeks to predict the "rating" or "preference" a user would give to an item using user data, such as the location and page view history, and utilizes the data to predict our interest in a specific item. For example, in an advertisement system, user interest is measured with click-through rate (CTR), which is the probability that we are actually going to click in and see more detail.

Content filtering was the most common technique used in early recommendation systems. A set of experts classified products into categories, while users selected their preferred categories and were matched based on their preferences. Later on, collaborative filtering is introduced in the recommendation system, where recommendations are based on past user behaviors, such as prior ratings given to products. Neighborhood methods that provide recommendations by grouping users and products together and latent factor methods that characterize users and products by certain implicit factors via matrix factorization techniques were later deployed with success.

Deep learning is one of the most exciting breakthroughs of artificial intelligence and is extensively applied to solve real-world problems in many areas such as speech recognition, computer vision, natural language processing, and medical diagnosis. Deep-learning-based recommendation models (DLRMs) are key tools in serving personalized content to users at Internet scale [8, 17, 27]. As the value generated by recommendations often relies on the system's ability to reflect recent data, production services frequently retrain DLRM on new data and roll out the newly-trained DLRMs into production [4]. Reducing the amount of time it takes to train a DLRM is thus critical to maintaining an accurate and up-to-date model.

Typically in recommendation models, training samples are extremely sparse, meaning the number of total features available is usually many scales greater than the number of features presented in each sample. For example, in current recommendation systems, petabytes of log data of user behavior are generated every day. Training samples typically contain billions to trillions of features, while only a few of these dimensions are non-zero for each sample.

To handle high-dimensional sparse training samples, DLRMs consist of embedding tables and neural networks. Embedding tables are large matrices that map sparse categorical features (e.g., properties of a user) to a learned dense representation [17]. Embedding tables can be thought of as lookup tables where rows (called "entries") correspond to sparse features (typically in millions or billions [12, 17]) and columns correspond to dense representations (typically in

Features	Samples per day	Average IDs/sample	Total model size
1 Billion	1.5 Billion	5000	17TB

Table 1.1: Alibaba's DLRM sizes.

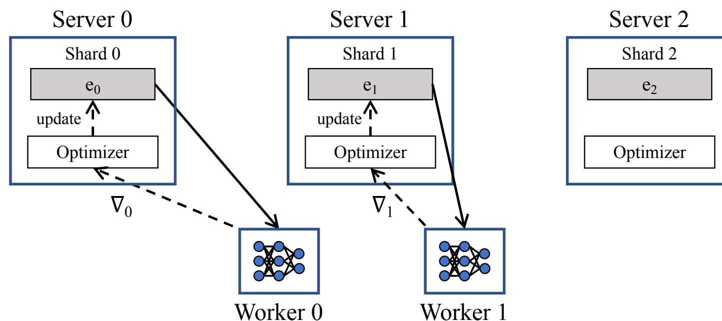

Figure 1.1: Example of the distributed setup used to train DLRMs.

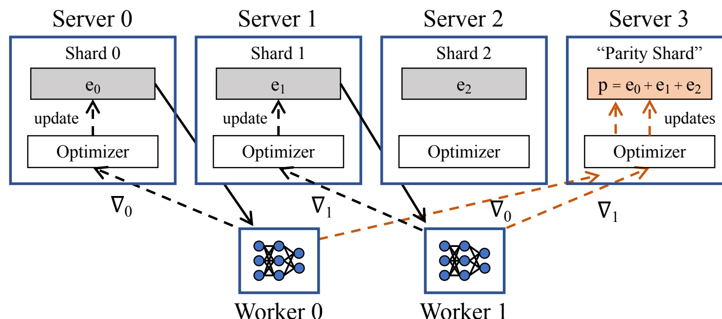

Figure 1.2: Naive erasure-coded DLRM with $k = 3$ and $r = 1$.

tens or hundreds). A small fully-connected neural network processes the dense representations corresponding to embedding table entries for a given training sample. Embedding tables are generally large, typically ranging from hundreds of gigabytes to terabytes in size [17]. In contrast, the neural networks used in DLRMs are comparatively smaller. Table 1.1 shows the typical volume of production data used by Alibaba's DLRM called XDL. It shows 17TB of model parameters needs to be stored in main memory.

The de facto approach to training such large models is to distribute training across a cluster of tens or hundreds of nodes [17], as depicted in Figure 1.1. Embedding tables and neural network parameters are sharded across a set of *servers* and kept in memory for fast access. *Workers* perform neural network training by accessing model parameters from servers and send gradients to servers to update parameters via an optimizer (e.g., Adam). In a single training iteration, a worker reads embedding table entries corresponding to the given training sample from servers, performs a forward and backward pass over the neural network using the retrieved entries to generate gradients, and sends gradients to the servers hosting the corresponding parameters. An optimizer (e.g., Adam) on each server calculates updates for model parameters based on the

received gradients and the optimizer's internal state, and applies updates to the corresponding parameters. The many workers in the system train in parallel, typically in an asynchronous fashion [17]. As each training sample accesses only a few of the billions of embedding table entries, embedding table entries are updated sparsely. In contrast, all neural network parameters are typically updated in each training iteration.

Training DLRMs is resource and time intensive, often taking multiple days or weeks. Since model parameters are stored in memory, any server failure requires training to restart from scratch. Given that failures are common in large-scale settings, it is imperative for DLRM training to be fault tolerant. Checkpointing is the predominant approach employed for fault tolerance in DLRM training [17]. This involves periodically pausing training and writing the current parameters and optimizer state to stable storage. If a failure occurs, the entire system resets to the most recent checkpoint and restarts training from that point.

While simple, checkpointing requires frequent pauses during training to write model state to stable storage and a lengthy recovery process to redo lost work after failure. We show in §2.2 these pauses can significantly increase training time, and that this overhead increases with DLRM size, causing 4%-33% training overhead during normal training even without any failure. Our analysis is in line with observations from Facebook in a recent concurrent study [24]. Given the common trend of increasing model size to improve accuracy [23, 31] checkpointing is slated to become an even larger overhead in training future DLRMs.

An alternative to checkpointing is to replicate DLRM state. In a replication-based DLRM training system, model parameters are replicated onto separate servers and gradients are sent to all servers containing replicas of the corresponding parameter. By maintaining multiple copies of up-to-date model parameters on separate servers, the system can immediately continue training in the event of a server failure. However, replication requires at least $2\times$ as much server memory as checkpointing. Given the large memory footprint of embedding tables even in the absence of redundancy, replicating embedding tables is impractical.

An ideal approach to fault-tolerant DLRM training would (1) operate with low training-time overhead during normal operation and recovery (like replication), with (2) low memory overhead (like checkpointing).

Erasure codes are coding-theoretic tools for adding proactive redundancy (like replication) but with significantly less memory overhead, which have been widely employed in storage and communication systems (e.g., [16, 29, 33, 34, 37]). An erasure code encodes k data units to generate r redundant "parity units" such that any k out of the total $(k + r)$ data and parity units are sufficient for a decoder to recover the original k data units. Therefore, erasure codes operate with resource overhead of $\frac{k+r}{k}$, which is less than that of replication by setting $r < k$. These properties have made erasure codes a widely-deployed alternative to replication in storage and communication systems [29, 34].

Due to their low overhead, erasure coding offers promising potential for imparting efficient fault tolerance to DLRM training. An example is demonstrated in Figure 1.2. In this example, a parity parameter p is constructed from parameters e_0, e_1, and e_2 via the encoding function $p = e_0 + e_1 + e_2$, and placed on a separate server. If a server fails, the system recovers lost parameters by reading the k available parameters and performing the erasure code's decoding process (e.g., $e_1 = p - e_0 - e_2$).

While erasure codes appear promising for imparting efficient fault tolerance to DLRM train-

ing, there are a number of challenges in bringing this vision to practice. (1) Parities must be kept up-to-date with their corresponding DLRM parameters to ensure correct recovery. This requires additional communication and computation in the system, which can reduce throughput. (2) As will be shown in §3.3.1, correctly updating parities when using optimizers that store internal state (e.g., Adagrad, Adam) is challenging without incurring significant memory overhead. (3) An erasure code's recovery process is typically resource intensive [33, 35]. This can potentially lead to long recovery times during which training can stall.

In this thesis, we present ECRM,[1] an erasure-coded DLRM training system that overcomes the aforementioned challenges through careful system design adapting simple erasure codes and ideas from storage systems to DLRM training. ECRM enables correct and low-overhead operation in the absence of failures (challenges 1 and 2) by delegating the responsibility of keeping parity entries up-to-date to servers, rather than workers. This maintains low training-time overhead, and circumvents the difficulty of maintaining correctness with stateful optimizers. ECRM recovers quickly from failure (challenge 3) by enabling training to continue during the erasure code's recovery process. The net result of ECRM's design is a DLRM training system that *recovers quickly from failures with low training-time and memory overhead, and without requiring pauses during training or recovery*.

We implement ECRM atop XDL, an open-source, industrial-scale DLRM training system developed by Alibaba [17]. We evaluate ECRM in training the DLRM used for the Criteo dataset [1] in MLPerf [2] and other variants across 20 nodes. ECRM recovers from failures significantly faster than checkpointing and operates with lower training-time overhead during normal operation. For example, ECRM reduces training-time overhead by up to 88% compared to checkpointing (more precisely, from 33.4% to 4%). ECRM's benefits in training-time overhead improve for larger DLRMs, showing the promise of ECRM in imparting efficient fault tolerance to the training of current and future DLRMs. Furthermore, ECRM recovers from failure up to 10.3× faster than the average case for checkpointing, and, critically, enables training to continue during recovery with only a 6%–12% drop in throughput, while checkpointing forces training to pause during recovery. ECRM's benefits come at the cost of additional memory requirements and load on the training cluster. However, ECRM keeps memory overhead to only a fractional amount and balances additional load evenly among servers. These results showcase the promise of erasure coding as an alternative to checkpointing to enable low-latency, resource-efficient fault tolerance to current and future DLRM training systems.

In this thesis, we make the following contributions:

- Analyzing the overhead of checkpointing in distributed DLRM training systems.
- Identifying the potential of using erasure-codes to impart low-overhead fault tolerance to DLRM training systems, as well as the challenges in doing so.
- Designing, implementing, and evaluating ECRM, the first erasure-coded DLRM training system, which overcomes the challenges in applying erasure coding to DLRM training.

[1] ECRM: Erasure-Coded Recommendation Model

Chapter 2

Background and Motivation

We next provide background on DLRM training systems and the inefficiency of current approaches to fault tolerance in such systems.

2.1 DLRM training systems

DLRMs are widely deployed at Internet scale to deliver personalized content to users [8, 17, 27]. These models take in as input a set of categorical features (e.g., about a user), and return a prediction (e.g., video or advertisement recommendation). DLRMs consist of two primary components: (1) embedding tables that translate categorical features into learned dense representations, and (2) a neural network that takes in the resultant dense representation to deliver a prediction. Embedding tables are typically massive in size, spanning hundreds of gigabytes to terabytes [17]. In contrast, the neural networks used are comparatively smaller, often consisting of a few fully-connected layers [27].

As described in §1, DLRMs are typically large in size due to embedding tables that span hundreds of gigabytes to terabytes in size, and DLRM training is typically distributed across a set of servers and workers (Figure 1.1). Consequently, model parameters are sharded across servers and kept in memory for fast access. In a training iteration, workers first read the embedding table entries in the batch of training samples and compute a dense representation with the embedding table entries. Next, the workers perform a forward and backward pass over the neural network using the dense representations computed as inputs. Using the gradients calculated during backward pass, each worker first updates neural network parameters locally, and sends embedding table gradients back to the servers hosting the entries. An optimizer (e.g., Adagrad) on each server uses received gradients to update model parameters via a so-called update function. We note that there are two methods widely used to stored neural network parameters: on parameter servers or on workers. In the first method, neural network parameters are stored on the parameters server, same as the embedding tables. In each training iteration, workers will pull the entire neural network from the parameter servers, and perform training locally. In the second approach, neural network parameters are replicated across workers. Therefore, the parameter updates in the backward pass are accumulated with an `allreduce` and applied to the replicated parameters on each device with a specific interval.

Each sample used in training typically accesses only a few embedding table entries, but all neural network parameters. Thus, embedding table entries are accessed and updated sparsely, while neural network parameters are updated frequently. Finally, many workers proceed in a data-parallel fashion, where each worker is pre-assigned a number of distinct training samples to train on. Many systems, such as those used by Facebook and Alibaba [17, 27], use asynchronous training, where each worker is assigned a number of batches of training samples, and proceed through the training samples without waiting for any other worker. Alternatively, in synchronous training, each worker works on one batch of training data, and proceed to the next batch only after all workers are done with the current batch. We focus on this asynchronous regime in this work, but describe in §3.5 how the techniques we propose can apply to synchronous training.

Many popular optimizers use per-parameter state in updating parameters (e.g., Adam, Adagrad, momentum SGD). We refer to such optimizers as "stateful optimizers." For example, Adagrad [10] tracks the sum of squared gradients for each parameter over time and uses this when updating the parameter. Per-parameter optimizer state is kept in memory alongside model parameters on servers and is updated when the corresponding parameter is updated. As per-parameter state grows with the number of DLRM parameters [31], optimizer state for embedding tables can consume a large amount of memory.

2.2 Checkpointing and its downsides

Given the large number of nodes on which DLRMs are trained, failures are to be expected during training [17]. Due to the time it takes to train such models and the fact that such model is usually retrained on a constant basis, it is critical that DLRM training be made fault tolerant so training progress won't be lost due to failure. Currently, checkpointing is the primary approach used to achieve fault tolerance in DLRM training. Under checkpointing, training is periodically paused and DLRM parameters and optimizer state are written to stable storage (often via a distributed file system, such as HDFS). Upon failure, the most recent checkpoint is read back from stable storage, and the entire system restarts training from this checkpoint.

Checkpointing can significantly extend training time due to two time penalties (1) during normal operation and (2) during recovery. We will discuss each of the downsides thoroughly in this section.

2.2.1 Time penalty during normal operation

We first analyze and evaluate the overhead incurred by checkpointing on training in the absence of failures. Consider a system in which checkpoints are taken every C_P time units, and for which it takes C_W time units to write a checkpoint to stable storage. In such a system, training is paused every C_W out of every $C_P + C_W$ time units, giving checkpointing an overhead during normal operation of $\frac{C_W}{C_P+C_W}$. Writing checkpoints to stable storage is a slow process given the large sizes of embedding tables, and training is paused during this time so to ensure the consistency of the saved models. Intuitively, the overhead of checkpointing on normal operation increases the longer it takes to write a checkpoint and the more frequently checkpoint.

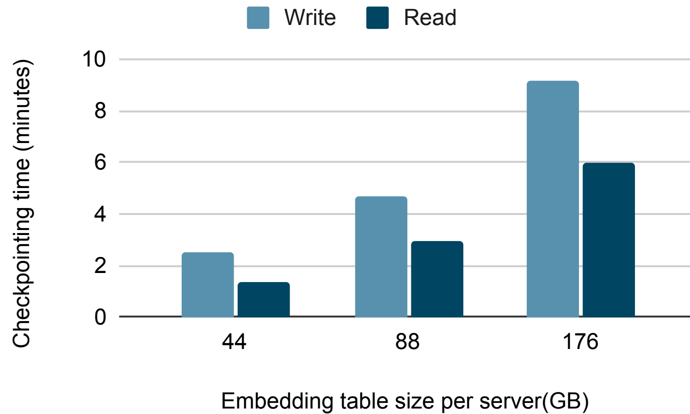

Figure 2.1: Time required to read and write checkpoints

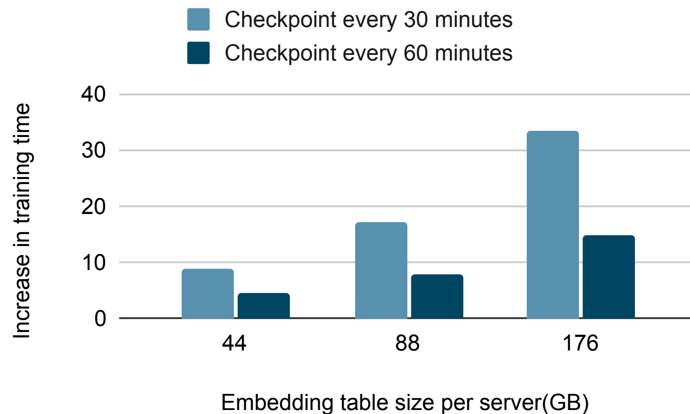

Figure 2.2: Effect of checkpointing on total training time

This mechanism that pauses training when checkpoints are being taken is commonly referred to as synchronous checkpointing. An alternative to synchronous checkpointing is asynchronous checkpointing where training resumes normally when a checkpoint is being taken. In asynchronous checkpointing, parameters can be updated during checkpointing and therefore can be inconsistent. As verified by our conversations with Facebook and Google's teams working on DLRM training, asynchronous checkpointing might have an unexpected effect on the convergence of the model. Therefore, synchronous checkpoints is the state-of-the-art approach to checkpointing DLRM systems and is most commonly adopted in the industry.

As described above, checkpointing frequently pausing training to save the current DLRM state to stable storage. To illustrate this overhead, we evaluate checkpointing DLRMs in XDL. Training is performed on a cluster of 15 workers and 5 servers, with checkpoints periodically written to an HDFS cluster (full setup described in §4.1). Production recommendation model training systems typically write checkpoints to general-purpose, HDFS-like distributed storage systems: Alibaba's recommendation model training system leverages HDFS, and a recent paper from Facebook [6] reports using their HDFS-based Hive storage system during training. We train the DLRM used for the Criteo Terabyte dataset in MLPerf and its variations, which requires 220-

880 GB of memory for embedding tables (44-176 GB per server), corresponding to memory sizes of 64 - 256 GB per server.

Figure 2.1 shows that the time overhead for writing checkpoints is significant (on the order of minutes. This overhead is inline with observations in production settings as confirmed by our discussions with multiple DLRM teams and a recent concurrent study by Facebook [24]. Figure 2.2 shows the overhead of checkpointing on normal training with two checkpointing periods: 30 and 60 minutes. We measure the time it takes for a each setup to reach the same number of iterations that a system with no fault tolerance (and thus no overhead) reaches in four hours. As expected, training time increases both with increased DLRM size and with decreased time between checkpoints.

2.2.2 Time penalty during recovery

Upon failure, checkpointing-based DLRM training systems must (1) roll back the DLRM to the state of the most recent checkpoint by reading the this checkpoint from stable storage and (2) redo any of the training iterations that occurred between the previous checkpoint and the failure. Training is paused during this time, as new training iterations cannot be completed.

Figure 2.1 shows that the time it takes to read back checkpoints from stable storage is significant and grows with DLRM size. In addition to the checkpoint reading time, the time required to redo lost training iterations depends on when failure occurs, which ranges from 0 to the checkpointing interval. For example, if checkpoints are written every C_P time units, this time will be zero in the best case (failing immediately after writing a checkpoint), C_P in the worst case (failing just before writing a checkpoint), and $\frac{C_P}{2}$ on average. Intuitively, increasing the time between checkpoints increases the expected recovery time.

Takeaways. Checkpointing suffers a fundamental tradeoff between training-time overhead in the absence of failures and when recovering from failure [9]. Increasing the time between checkpoints reduces the fraction of time paused saving checkpoints, but increases the expected amount of work to be redone upon recovery. Furthermore, the experiments above illustrate that time overheads both during normal operation and during recovery increase with increasing model size. Given the common trend of increasing model size to improve accuracy [23, 31] *checkpointing is slated to become an even larger overhead in training future DLRMs.* This calls for alternate approaches to fault tolerance in DLRM training.

2.3 Fault tolerance via proactive redundancy?

2.3.1 Replication

An alternative to checkpointing is to proactively provision redundant servers that can immediately take over for failed servers. Replication is the most common form of proactive redundancy. Replication for DLRM training would involve using twice as much memory to store copies of DLRM parameters and optimizer state on two servers. Gradients for a given parameter are sent to and applied on both servers holding copies of the parameter. The system seamlessly continues training if a single server fails by accessing parameters from the replica. Thus, replication allows

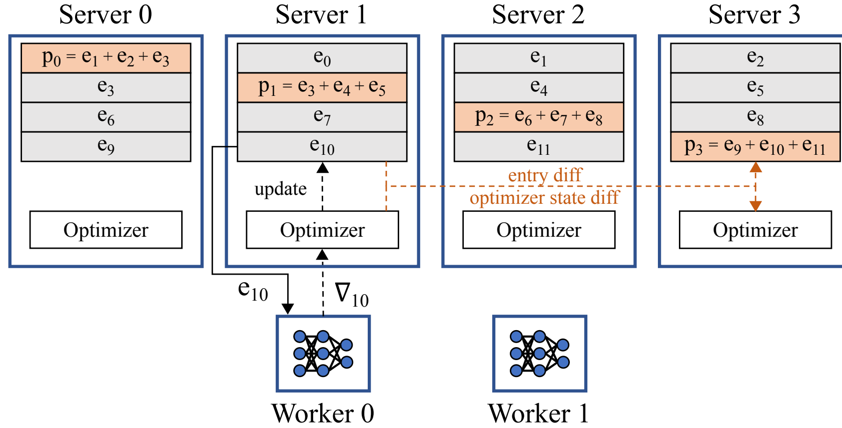

Figure 2.3: Example of ECRM with $k = 3$, $r = 1$.

training to proceed unscathed from failure. Replication successfully reduces the need for any rollback once failure occurs. Additionally, replication removes the overhead of pausing training due to synchronous checkpointing. However, a replicated DLRM training system requires at least twice as much memory as a non-replicated one. Given the large sizes of embedding tables, the memory overhead of replication is impractical for DLRM training systems.

Takeaways. Summarizing the advantages and disadvantages of checkpointing and replication, an ideal approach to fault-tolerant DLRM training would have (1) the low-latency recovery of replication and (2) the low memory overhead of checkpointing.

2.3.2 Erasure codes: proactive, low-overhead

Erasure codes are coding-theoretic tools used for imparting resilience against unavailability in storage and communication systems with significantly less overhead than replication [29, 34, 37]. An erasure code encodes k data units to generate r redundant "parity units" such that any k out of the total $(k + r)$ data and parity units suffice for a decoder to recover the original k data units. Therefore, erasure codes operate with overhead of $\frac{k+r}{k}$, which is less than that of replication by setting $r < k$. Figure 1.2 shows an example of how erasure codes could potentially be used in DLRM training. These properties have led to wide adoption of erasure codes in storage and communication systems [29, 34]. Due to the above reasons, we believe that erasure codes offer promising potential for achieving these goals to impart efficient fault tolerance to DLRM training. This thesis explores the potential of the use of erasure codes in DLRM training, unearthing the challenges and designing a system that overcomes them.

Chapter 3

ECRM: erasure-coded training

We now describe ECRM, a system that imparts efficient fault tolerance to DLRM training through careful system design adapting simple erasure codes and ideas from storage systems to DLRM training. Using erasure codes in DLRM training raises unique challenges compared to the traditional use of erasure codes in storage and communication. We first provide a high-level overview of ECRM and then discuss these challenges and how ECRM overcomes them.

3.1 Overview of ECRM

Figure 2.3 provides a high-level picture of erasure-coded operation in ECRM. ECRM encodes DLRM parameters using an erasure code and distributes the resultant parities throughout the cluster before training begins. Groups of k embedding table entries from separate servers are encoded together to produce r parities that are stored in memory on separate servers. ECRM thus requires $\frac{k+r}{k}$-times as much memory as the original system. We describe in §3.2 exactly which parameters are encoded and how parities are placed throughout the cluster. As encoded parameters are updated during training, ECRM must also keep the corresponding parities up-to-date. In the event of a server failure, ECRM uses the erasure code's decoder to reconstruct lost DLRM parameters.

While the use of erasure codes in DLRM training is enticing, there are many system design decisions and challenges that affect the correctness and efficiency of erasure-coded DLRM training: (1) Which parameters of a DLRM should be encoded and where should parities be placed (§3.2)? (2) How can parities be updated correctly and efficiently (§3.3)? (3) How can ECRM avoid pausing training when recovering from failure (§3.4)? (4) How can ECRM guarantee the consistency of the DLRM recovered after failure (§3.5)?

We next describe how ECRM addresses these system design choices and challenges. Figure 3.1 illustrates the components added to servers in ECRM that will be described next for maintaining correct and efficient operation for reference in future sections.

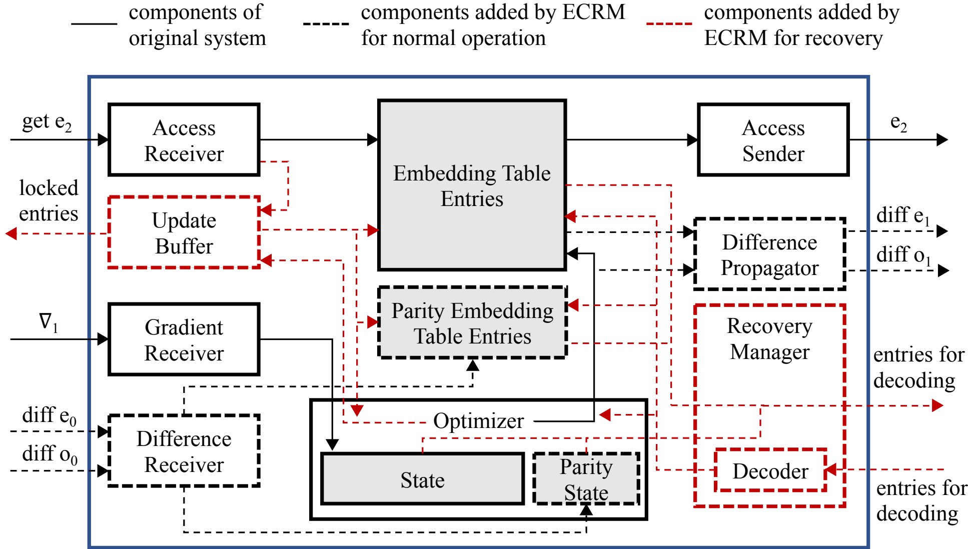

Figure 3.1: Components and operation of a server in ECRM. Shaded boxes store data, and unshaded boxes are used for control flow.

3.2 Encoding and placing parity parameters

DLRMs have many parameters: embedding tables, neural networks, and optimizer state. We next describe how ECRM selects which parameters should be encoded and where in the cluster the resultant parities should be placed.

Which parameters should be encoded? Fault tolerance is primarily needed in DLRM training to recover failed servers, which hold DLRM parameters and optimizer state. If a server fails, the portion of the DLRM hosted on that server is lost, and training cannot proceed. In contrast, DLRM training systems with architectures as described in §2.1 are naturally tolerant of worker failures, as the system can continue training with fewer workers while replacement workers are provisioned.

Furthermore, as each worker pulls all neural network parameters from servers when training, the neural network is naturally replicated on workers. If a server fails, the neural network parameters it held can be recovered from a worker.[1]

In contrast, embedding tables and optimizer state are *not* naturally replicated. Embedding tables and optimizer state are sharded across many servers, and each worker accesses only a few entries in each training iteration. Thus, lost embedding table entries and optimizer state cannot be recovered from workers. Furthermore, replicating embedding tables and optimizer state is impractical, given their large size. Thus, ECRM encodes only embedding tables and optimizer state; neural network parameters need not be encoded.

Where should parities be placed? Recall from §2.1 that embedding tables and optimizer

[1]While the asynchronous training described in §2.1 does not guarantee that all workers will have the most up-to-date neural network parameters, recovering neural network parameters from a worker will still result in recovering a neural network that is equivalent to one that could be observed under asynchronous training.

state are sharded across servers. ECRM encodes groups of k embedding table entries from different shards to produce a "parity entry," and places the parity entry on a separate server. Optimizer state is also encoded to form "parity optimizer state," which is placed on the same server hosting the corresponding parity entry.

The parity entries in ECRM are updated whenever any of the k corresponding embedding table entries are updated. Hence parity entries are updated significantly more frequently than the original embedding table entries. Parities must be placed carefully within the cluster so as not to introduce load imbalance among servers for updating parities. ECRM uses rotating parity placement to distribute parities among servers, resulting in an equal number of parities per server. An example of this approach is illustrated in Figure 2.3 with $k = 3$. Each server is chosen to host a parity in a rotating fashion and the entries used to encode that parity are hosted on the 3 other servers in the system. This approach is inspired by the approach of placing parities in RAID-5 [29] hard-disk systems.

Encoder, decoder, and sharding. Embedding tables and optimizer state are encoded and distributed throughout the cluster prior to beginning training. During encoding, each embedding table is divided into groups of k embedding table entries. Groups of k embedding table entries from different shards are then encoded together to produce r redundant "parity entries," and all $(k + r)$ entries are placed in memory on separate servers. If the training utilizes a stateful optimizer, the optimizer state corresponding to each embedding table entry is also encoded to form "parity optimizer state," which is placed on the same server hosting the corresponding parity entry. ECRM thus requires $\frac{k+r}{k}$-times as much memory as the original training system. This can be accomplished by either using more memory per server, or by provisioning $\frac{k+r}{k}$-times as many servers.

We focus on using erasure codes with parameter $r = 1$ (i.e., constructing a single parity from k embedding table entries and being able to recover from a single failure) throughout this work. Within this setting, ECRM uses the simple summation encoder illustrated in Figure 2.3, and the corresponding subtraction decoder. For example, with $k = 3$, embedding table entries e_0, e_1, and e_2 are encoded to generate parity p as $p = e_0 + e_1 + e_2$. If the server holding e_1 fails, e_1 will be reconstructed as $e_1 = p - e_0 - e_2$. We focus on this $r = 1$ for a few reasons:

1. $r = 1$ represents the most common failure scenario experienced by a cluster in datacenters [32, 33].
2. The unlikely event of more than one failure among $k + 1$ servers happening at a time is not catastrophic in ECRM, as it simply requires restarting training.

Though ECRM currently focuses on recovering from a single failure, it can easily be adapted to cases in which higher fault tolerance is merited with $r > 1$. Currently in ECRM given a coding scheme with parameters r and k, any $r + 1$ simultaneous server failures among all servers could cause the system to be unable to recover. The likelihood of such failure increases with the number of servers. To reduce the likelihood of such events, ECRM can be adapted to leverage "coding groups." A coding group is a group of $k + r$ servers where all parameters stored on any server in the group is only coded with parameters from other servers in the same group. ECRM divides servers into coding groups of size near $k+r$, and place parity entries correspondingly. To cause a failure with such a system using coding groups, $r+1$ server failures must happen simultaneously in the same coding group of $k + r$. The likelihood is much lower and is independent of the total

number of servers.

As we pointed out, it's high unlikely that two servers failed within the same coding group and cause ECRM to be unable to recover. Even though it's highly unlikely, we want to point out that ECRM can utilize multi-level checkpointing [26] with a much lower checkpointing frequency as a backup. Modern DLRMs are retrained constantly from a daily basis. After a longer time interval, DLRMs have to be written to stable storage, regardless of fault tolerance. Such low checkpoint frequency will serve as the backup recovery solution given at highly unlikely failure of two server simultaneously, and allows training to restart from a reasonable point.

3.3 Correctly and efficiently updating parities

As described in §3.2, ECRM must keep parity entries up-to-date to enable an erasure code to correctly reconstruct lost embedding table entries. We now describe challenges with keeping parity entries up-to-date and how ECRM overcomes them.

3.3.1 Challenges in keeping up-to-date parities

Maintaining correctness with stateful optimizers. Embedding table entries are updated when workers send a set of gradients corresponding to the corresponding embedding table entries at a server. As described in §3.2, ECRM maintains a single parity entry that is the sum of k embedding table entries. To maintain this invariant, ECRM needs to guarantee that the parity entries are updated correctly to be the sum of the embedding table entries after each gradient update to one of the k entries throughout training.

To illustrate the challenges with keeping parity entries up-to-date, we will first illustrate how a naive approach to erasure-coded DLRM training would keep parities up-to-date. First, consider the SGD update function in which parameter e_0 is to be updated using gradient ∇_0. Let $e_{i,t}$ denote the value of embedding table entry e_i after t updates, and $\nabla_{i,t}$ denote the gradient for $e_{i,t}$. SGD updates e_0 using learning rate α as:

$$e_{0,t+1} = e_{0,t} - \alpha \nabla_{0,t} \tag{3.1}$$

A closer look at the properties of this update function illustrates that parity p can be kept up-to-date by simply applying the same update using gradient ∇_0 *directly on the parity, without accessing other embedding table entries*:

$$\begin{aligned} p_{t+1} &= p_t - \alpha \nabla_{0,t} && (3.2)\\ &= (e_{0,t} + e_{1,t} + e_{2,t}) - \alpha \nabla_{0,t} && (3.3)\\ &= (e_{0,t} - \alpha \nabla_{0,t}) + e_{1,t} + e_{2,t} && (3.4)\\ &= e_{0,t+1} + e_{1,t} + e_{2,t} && (3.5) \end{aligned}$$

The same argument holds for all linear update functions applied atop a linearly-encoded parity.

However, this naive approach to erasure-coded DLRM training suffers a fundamental challenge in *correctly* updating parity entries when using a stateful optimizer. Consider the same

example described above but now using the Adagrad optimizer [10] instead of SGD. The update performed by Adagrad for $e_{0,t}$ with gradient $\nabla_{0,t}$ is:

$$e_{0,t+1} = e_{0,t} - \frac{\alpha}{\sqrt{G_{0,t} + \epsilon}} \nabla_{0,t} \tag{3.6}$$

where α is a constant learning rate,

$$G_{0,t} = \nabla_{0,0}^2 + \nabla_{0,1}^2 + \ldots + \nabla_{0,t}^2 \tag{3.7}$$

is the sum of squares of the previous gradients for parameter e_0, and ϵ is a small constant. $G_{0,t}$, which we call e_0's "accumulator," is an example of optimizer state.

As described in §3.2, ECRM maintains one "parity accumulator" per parity entry. For example, using the encoder described in §3.1, a parity accumulator for this example would be $G_p = G_0 + G_1 + G_2$. This parity accumulator is easily kept up-to-date by adding the squared gradient for updated entries to the parity accumulator. However, using this parity accumulator to update the parity entry based on $\nabla_{0,t}$ would result in an incorrect parity entry, as $G_{0,t} \neq G_{p,t}$.

This issue arises for any stateful optimizer, such as Adagrad, Adam, and momentum SGD. Given the popularity of such optimizers, ECRM must employ some means of maintaining correct parities when using stateful optimizers.

One potential approach to overcome this issue is by keeping replicas of the optimizer state of each of the k embedding table entries corresponding to the parity on the server hosting the parity. However, as described in §3.1, optimizer state is typically large and grows in size with embedding tables. Thus, replicating optimizer state is impractical.

Maintaining low overhead in the absence of failures. Even if the issues described above were not present, the naive approach to erasure-coded DLRM training shown in Figure 1.2 will have high training-time overhead. Under this naive approach, keeping parity entries up-to-date requires that gradients for a given embedding table entry be communicated both to the server hosting the entry as well as to the server hosting the corresponding parity entry, and that the optimizer's update function be applied on both servers. Thus, maintaining up-to-date parity entries can result in overhead in network bandwidth and compute for workers. Given that workers are typically the bottleneck in DLRM training systems [17], ECRM must minimize the effect of this overhead on training throughput.

3.3.2 Difference propagation

The challenges described above stem from sending gradients directly to the servers hosting parities, a naive approach which we term "gradient propagation." Under gradient propagation, workers must do additional work to send duplicate gradients, resulting in CPU and network bandwidth overhead on workers. Servers holding parity entries receive only the gradient corresponding to the original embedding table entry and must both calculate an optimizer's update function and correctly update the parity entry and optimizer state. As described above, performing these updates correctly given only parity optimizer state and gradients is challenging.

To overcome these downsides, ECRM introduces *difference propagation*. As illustrated in Figure 2.3, under difference propagation, workers send gradients only to the servers holding embedding table entries corresponding to that gradient. After applying the optimizer's update function to embedding table entries and updating optimizer state, the server then asynchronously sends the *differences* in the entry and optimizer state to the server holding the corresponding parity entry. The receiving server adds these differences to the corresponding parity entry and optimizer state. Note that we use a linear encoder for the parity entries, so the coding will be automatically maintained by sending and updating difference.

Difference propagation has three key benefits over gradient propagation.

1. By sending differences to servers, rather than gradients, difference propagation updates parity entries correctly when using stateful optimizers.
2. Difference propagation adds no overhead to workers. This is important, given that workers are typically the bottleneck in DLRM training [17].
3. Parity updates can be performed asynchronously, and potentially lazily with no urgency, which allows better utilization of servers' resources advantages of difference propagation over the naive approach.
4. Difference propagation avoids computing the optimizer's update function on both the server holding the original embedding table entry and the server holding the parity entry, as is required in gradient propagation. This saves server CPU cycles.

Difference propagation does introduce network and CPU overhead on servers for transmitting and applying differences. This overhead grows with the amount of state used by an optimizer. Despite this, §4 will show that difference propagation significantly outperforms gradient propagation.

3.4 Pause-free recovery from failure

We next describe how ECRM recovers from failure without requiring training to pause.

ECRM inherits XDL's approach for detecting server failures: one worker is delegated as the coordinator, and all servers periodically send heartbeat messages to the coordinator. If a heartbeat message is missed from a server, the server is considered to have failed, and the coordinator triggers recovery. XDL uses a ten second heartbeat interval by default. While this leaves a window of time from when a server has failed to when recovery is triggered, all workers that attempt to contact the failed server will block until recovery takes place. Thus, new training iterations will not begin after the server has failed.

Once a failure is detected, all workers stop training new data batches and attempt to finish sending all gradients that have been already calculated. After all workers receive either acknowledgements or failure messages regarding the gradient updates and the failed server restarts, the recovery process begins. Due to the property of the erasure codes described in §2.3 that any k out of the total $(k+1)$original and parity units suffice to recover the original k units, ECRM can continue training even when a single server fails. For example, a worker in ECRM could read entry e_1 in Figure 2.3 even if Server 2 fails by reading e_0, e_2, and p, and decoding $e_1 = p - e_0 - e_2$.

Reading unavailable data in such a manner is commonly referred to as operating in "degraded mode" in erasure-coded storage systems.

3.4.1 Challenges in erasure-coded recovery

Despite the ability to perform degraded reads, ECRM must still fully recover failed servers to remain tolerant of future failures. However, prior work on erasure-coded storage has shown that full recovery can be time-intensive [33, 35]. Full recovery in ECRM requires reconstructing all embedding table entries and optimizer state held by the failed server. Given the large sizes of embedding tables and optimizer state, waiting for full recovery to complete before resuming training can significantly pause training. Thus, waiting for full reconstruction of a failed server before continuing to train can delay training for a significant period of time.

3.4.2 Training during recovery in ECRM

Rather than solely performing degraded reads after a failure or pausing until full recovery is complete, ECRM *enables training to continue while full recovery takes place.* Upon failure, ECRM begins full recovery of lost embedding table entries and optimizer state. In the meantime, the system continues performing new training iterations, with workers performing degraded reads to access entries from the failed server. If a worker needs to read an embedding table entry from the failed server, it does so via a degraded read by reading k embedding table entries from the k other servers encoded with the missing entry, and decoding the needed embedding table entry on demand.

Care must be taken to ensure correct recovery when performing new training updates concurrently with full recovery. In particular, ECRM must avoid updating an embedding table entry in parallel with its use for recovery. If the recovery process reads the new value of the entry, but the old value of the parity entry (e.g., because the update was not yet applied to the parity), then the recovered entry will be incorrect. (see §3.5 for an example).

To ensure correctness of the recovered embedding tables, ECRM employs granular locking to avoid such race conditions. At the beginning of the process, ECRM divides the embedding table in L equally-sized partitions. Each server initializes an empty write buffer and "locks" the first partition of the lost embedding table entries that the recovery process will decode. While the recovery process holds this lock, all updates to embedding table and parity entries that will be used in recovery for the locked partition are written to the write buffers on servers until the lock is released. Workers attempting to read an updated, but locked entry will do so by reading from the write buffer. When a lock is released, all buffered updates are applied to the original embedding tables, and the the lock will be switched to the next partition. The process will be repeated for all L partitions.

The number of embedding table entries covered by each lock introduces a tradeoff between time overhead in switching locks and server memory overhead for buffering updates. Increasing the number of locks will reduce the memory overhead due to the need to buffer fewer writes the expense of higher overhead in switching locks.We will demonstrate this tradeoff in §4.

There are various implementation of the write buffer. We choose to use an array implementation for the write buffer. Each server initializes an array, with equal size to one partition of the

embedding table stored on the server. Before the recovery of a partition of the embedding table, the server copies the entire embedding table partitions to the write buffer array. During recovery, all worker reads and writes are directly performed on the write buffer. The server flushes the write buffer by performing a single memory copy from the write buffer to the corresponding embedding table offset. The array implementation creates minimal overhead for the worker reads/writes since all embedding table entries can be accessed with a direct access. The array implementation also creates low overhead at lock switching by performing a single memory copy. While the array implementation creates a constant memory overhead, that is the worst case with the hashmap implementation, the memory overhead is strictly $1/L$ of the embedding table size, and can be alleviated with a larger number of locks.

3.5 Maintaining consistency of recovered DLRM

We next describe how ECRM provides the same guarantees regarding the consistency of a recovered DLRM as the general asynchronous training on top of which ECRM is built.

Consistency of individual parameters. ECRM ensures that each embedding table entry and optimizer state entry is recovered to the value from its most recent update that was applied both to the original entry and the parity. There is one case that requires care: when recovery is triggered while updating both an embedding table entry and its corresponding parity. If recovery is triggered after the update had been applied to the embedding table entry but before it has been applied to the parity entry, the decoded entry will be incorrect. ECRM avoids this scenario by ensuring that all in-flight updates are completed before recovery begins. As XDL ensures that the transmission and application of updates do not fail, this condition above is sufficient to guarantee the consistency of individual parameters.

Consistency across parameters. ECRM guarantees that a recovered DLRM represents one that could have been reached by asynchronous training, but does not guarantee that the recovered DLRM represents a state that was truly experienced during recovery. We will next illustrate this by example and show how the guarantee above results in ECRM providing the same consistency semantics as asynchronous training.

Consider the following timeline of events in DLRM training with embedding table entries x and y. We consider the state of the DLRM to be the combined state of each of these parameters.

As illustrated in the Table 3.1, due to the asynchronous property of difference propagation, the recovery process results in a DLRM state $\{x_t, y_{t+1}\}$ that was never experienced during training: in training, x was in state $t+1$ before y_t was even read.

Though the DLRM state recovered by ECRM in the timeline above was never truly experienced during training, it *is a DLRM state that could have just as easily been experienced during asynchronous training*. Under asynchronous training, it would be just as valid for the event at time 0 to have been performed after the event at time 2, which would have resulted in the DLRM state being $\{x_t, y_{t+1}\}$ for a period. Thus, the state recovered by ECRM is still valid from the lens of asynchronous training.

Time	Prev. State	New State	Event
0	x_t, y_t	x_{t+1}, y_t	Embedding table entry x is updated from x_t to x_{t+1} on Server 0. Entry and optimizer difference is asynchronously propagated to the server holding the parity.
1	x_{t+1}, y_t	x_{t+1}, y_t	Embedding table entry y_t is read from Server 1.
2	x_{t+1}, y_t	x_{t+1}, y_{t+1}	Embedding table entry y is updated from y_t to y_{t+1} on Server 1. Entry and optimizer difference is asynchronously propagated to the server holding the parity.
3	x_{t+1}, y_{t+1}	x_{t+1}, y_{t+1}	The parity corresponding to entry y is updated to reflect the update to y.
4	x_{t+1}, y_{t+1}	x_{t+1}, y_{t+1}	Server 0 fails, having not yet transmitted the difference for x.
5	x_{t+1}, y_{t+1}	x_t, y_{t+1}	The recovery process decodes x.

Table 3.1: Example timeline that results in ECRM inconsistency.

ECRM in synchronous training settings. As described in §2.1, many of the organizations deploying some of the mostly widely used recommendation systems use asynchronous training [17, 27]. As described in §4.1, we build ECRM atop XDL, an asynchronous training framework from Alibaba. However, ECRM can also support synchronous training. Synchronous training adds a barrier after certain number of training iterations in which workers communicate gradients with one another and servers, combine these gradients, and perform a single update to each modified parameter. In such a synchronous framework, ECRM would require that parity entries also be updated during this barrier so that they are kept consistent with training updates. As this setting is not the focus of our work, we leave a full study and evaluation of ECRM in synchronous settings to future work.

3.6 Tradeoffs in ECRM

We next discuss the effect of parameter k in ECRM as well as the consistency guarantees that can be made by ECRM.

Recall from §3.1 that ECRM encodes k embedding table entries into a single parity entry ($r = 1$) (and similarly for optimizer state). The parameter k results in tradeoffs in resource and time overhead and fault tolerance in ECRM, some of which differ significantly from traditional use of erasure codes.

Increasing k decreases fault tolerance. As ECRM encodes one parity entry for every k embedding table entries (same for optimizer state), since the erasure codes employed by ECRM can

recover from any one out of $(k+1)$ failures, increasing k decreases the fraction of failed servers ECRM can tolerate.

Increasing k decreases memory overhead. ECRM encodes one parity entry for every k embedding table entries (same for optimizer state). ECRM thus requires less memory for storing parities with increased k.

Increasing k *does not* change load during normal operation. As each embedding table entry in ECRM is encoded to produce a single parity entry, each update applied to an entry will also be applied to one parity entry. Thus, the overall increase in load due to ECRM is $2\times$, regardless of the value of parameter k. In addition to this constant increase in load, we will also show in §4.3 that ECRM balances this load evenly with various values of k.

Increasing k increases the time to fully recover. Recovery in ECRM requires reading k available entries from separate servers and decoding (and similarly for optimizer state). Thus, the amount of network traffic and computation required during recovery increases with k, which increases the time it takes to fully recover a failed server. However, as described in §3.4.2, ECRM allows training to continue during this time.

Chapter 4

Evaluation

In this chapter, we evaluate the performance of ECRM. The highlights of the evaluation include:

- ECRM recovers from failure up to 10.3× faster than the average recovery time for checkpointing.
- ECRM enables training to proceed with only a 6%–12% throughput drop during recovery, whereas checkpointing requires training to completely pause.
- ECRM reduces training-time overhead by up to 88% compared to checkpointing (more precisely, from 33.4% to 4%). ECRM's improvements increase with increasing DLRM size, showing promise for training both current and future DLRMs.
- The increased load introduced by ECRM for updating parities is alleviated by improved cluster load balance, which helps reduce training-time overhead.

4.1 Evaluation setup

We implement ECRM in C++ on XDL, an open-source DLRM training system from Alibaba [17].

Dataset. We evaluate with the Criteo Terabyte dataset, which is used in MLPerf. We randomly draw from the dataset a number of examples equivalent to one day of the dataset by picking each sample with a fixed probability $\frac{1}{24}$ in one pass through the entire dataset, and use this subset in evaluation to reduce storage requirements. The random sampling described above ensures that the sampled dataset mimics the full dataset.

Models. We use the open-source DLRM architecture for the Criteo dataset used in MLPerf [27] and its variants. This DLRM has 13 embedding tables, for a total of nearly 200 million embedding table entries. Each entry maps to 128 dense features. We use SGD with momentum as the optimizer, which adds a single floating point value of optimizer state per parameter. Any other optimizer can similarly be handled. The total size of the embedding tables and optimizer state is 220 GB. The DLRM uses a multilayer perceptron with seven layers with 128–1024 features per layer as a neural network [3].

We evaluate on DLRMs of different sizes by varying the sizes of embedding tables in two ways: (1) Increasing the number of embedding table entries (i.e., sparse dimension). This requires more memory per server and increases the amount of data that must be checkpointed/erasure-coded and recovered, but does not change other resource consumption in the system. (2) Keeping same number of entries, but increasing the size of each entry (i.e., dense dimension). This increases the memory consumed per server, the amount of data that must be checkpointed/erasure-coded, and also other resource consumption during training: increasing the size of each entry increases the network bandwidth consumed in transferring entries and their gradients, the work performed by neural networks (as neural networks process entries), and the work done by servers in updating entries. We consider three variants of the DLRM: (1) Criteo-Original, the original Criteo DLRM, (2) Criteo-2S, which has $2\times$ the number of embedding table entries (i.e., $2\times$ the sparse dimension), and (3) Criteo-2S-2D, which has $2\times$ the number of entries and with each entry being $2\times$ as large (i.e., $2\times$ the sparse and dense dimensions). These variants have size 220 GB, 440 GB, and 880 GB, respectively.

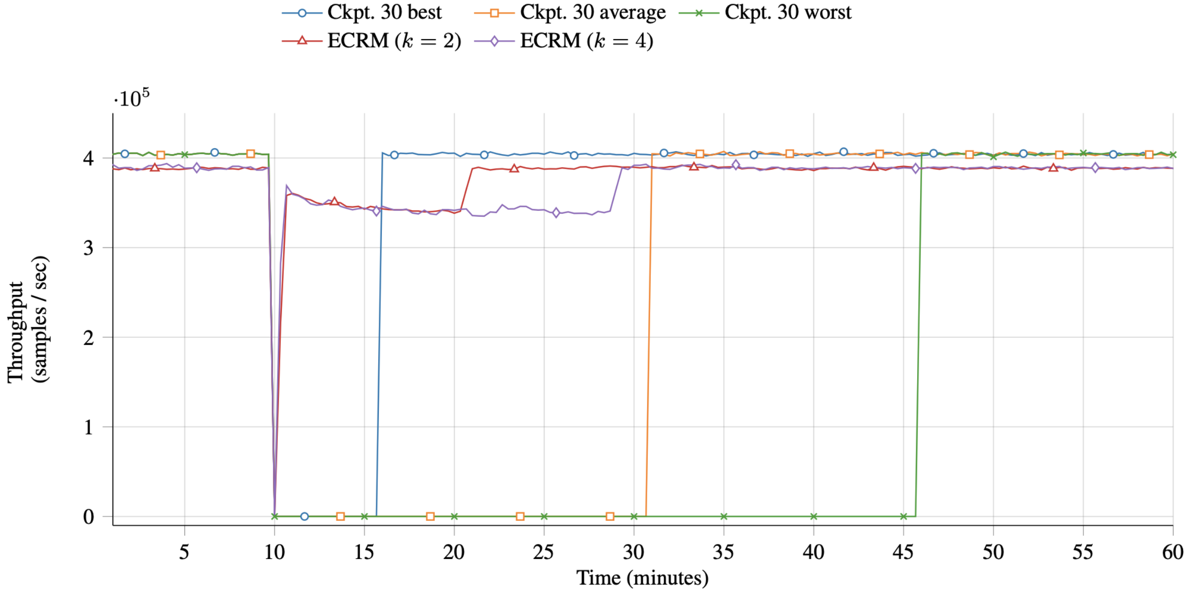

Figure 4.1: Throughput when recovering from failure at 10 minutes.

Coding parameters and baselines. We evaluate ECRM with $r = 1$ and k of 2, 4, and 10, representing scenarios with 50%, 25%, and 10% memory overhead, respectively. We compare ECRM to taking checkpoints to HDFS with every 30 minutes (Ckpt. 30) and every 60 minutes (Ckpt. 60), as production recommendation systems typically use general-purpose, HDFS like distributed storage systems. We evaluate with $k = 10$ in only a limited set of experiments due to the cost of the large cluster needed.

Cluster setup. We evaluate on AWS with 5 servers of type r5n.8xlarge, each containing 32 vCPUs, 256 GB of memory, and 25 Gbps network bandwidth (r5n.12xlarge is used for Criteo-2S-2D due to memory requirements). We use 15 workers of type p3.2xlarge, each equipped

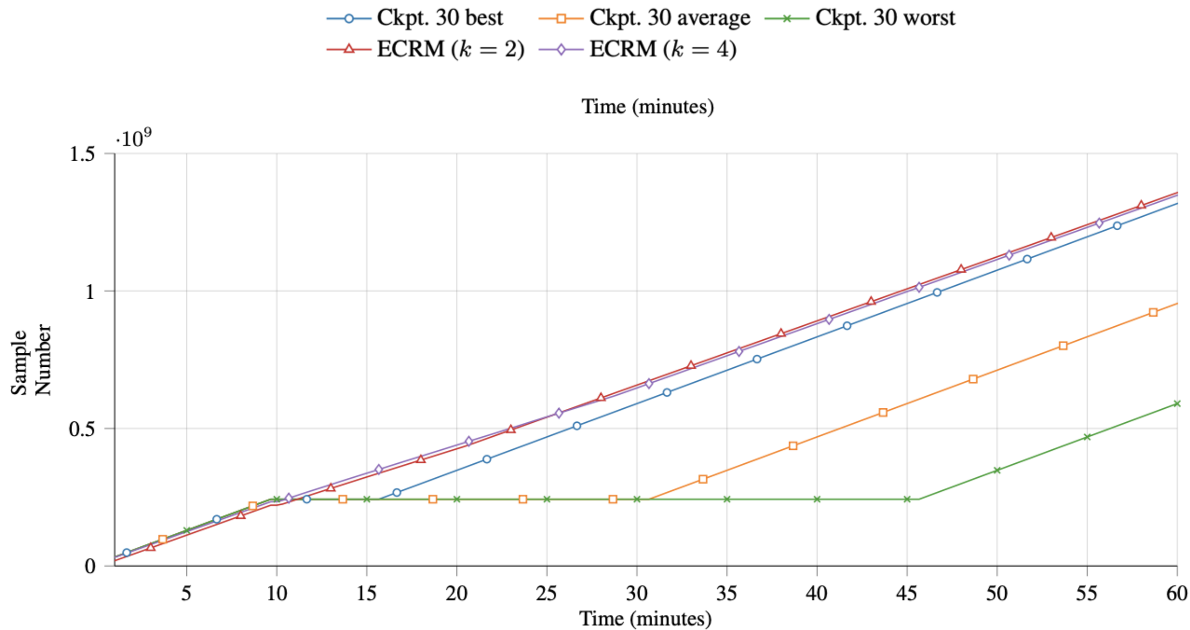

Figure 4.2: Training progress (bottom) when recovering from failure at 10 minutes.

with a V100 GPU, 8 vCPUs, and 10 Gbps of network bandwidth. This ratio of worker to server nodes is inspired from XDL [17]. We also evaluate with varying number of workers ranging up to 25 in §4.3. Each worker uses a batch size of 2048. When evaluating checkpointing, we use 15 additional nodes of type i3en.xlarge as HDFS nodes, each equipped with NVMe SSDs and 25 Gbps of network bandwidth. All nodes use AWS ENA networking. We perform additional experiments in which we limit the CPU and network resources available on servers to stress the overhead of ECRM's components.

Metrics. For performance during recovery, we measure the time to fully recover a failed server and the training throughput (in samples per second) during recovery. For performance during normal operation, we measure training-time overhead as percentage increase in the time to perform training on a certain number of samples and the training throughput (in samples per second).

4.2 Performance during recovery

We first evaluate ECRM and checkpointing in recovering from failure. As the recovery time for checkpointing depends on when failure occurs (see §2.2), we show the best-, average-, and worst-case recovery for checkpointing. Additionally, we compare the performance of ECRM recovery with different number of granular locks and evaluate its effect on the overall recovery performance.

The recovery performance of each approach is best illustrated in Figure 4.1, which shows

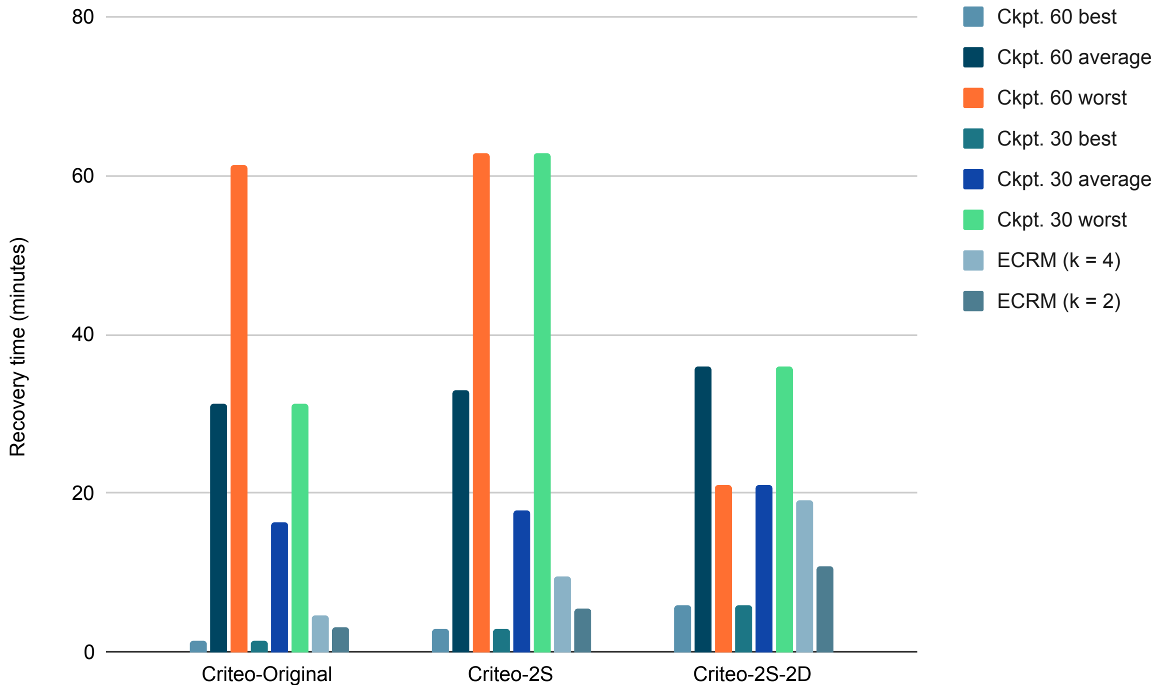

Figure 4.3: Time to fully recover a failed server.

the throughput and training progress of ECRM and Ckpt. 30 on Criteo-2S-2D after a single server failure (at time 10). ECRM fully recovers from the failure faster than the average case for Ckpt. 30, and, critically, maintains throughput within 6%–12% of that during normal operation during this time. As illustrated in the bottom figure, which plots the time taken to reach a particular number of training samples, ECRM's high throughput during recovery enables it to make greater progress in training than even the best case for Ckpt. 30. The recovery performance of Ckpt. 60 would have been even worse than that for Ckpt. 30, though we omit it from the plots for clarity.

Figure 4.3 shows the time it takes for ECRM, Ckpt. 30, and Ckpt. 60 to recover a failed server. ECRM recovers a failed server significantly faster than the average case of checkpointing. For example, ECRM with $k = 4$ recovers 1.9–6.8$\times$ faster and 1.1–3.5$\times$ faster than the average case for Ckpt. 60 and Ckpt. 30, respectively (and up to 10.3$\times$ faster with $k = 2$). While Ckpt. 30 does recover faster from failure than Ckpt. 60, §4.3 will show that Ckpt. 30 has significantly higher training-time overhead during normal operation. More importantly, unlike checkpointing, ECRM enables training to continue during recovery with high throughput.

Effect of parameter k. Figure 4.3 illustrates that it takes longer for ECRM to fully recover with higher value of parameter k. The intuition behind this is described in §3.6 However, Figure 4.1 shows that ECRM maintains high throughput during recovery for each value of k.

Effect of DLRM size. Figure 4.3 also shows that the time to fully recover increases with DLRM size for both ECRM and checkpointing, as expected (see §2.2 and §3.6). ECRM's recov-

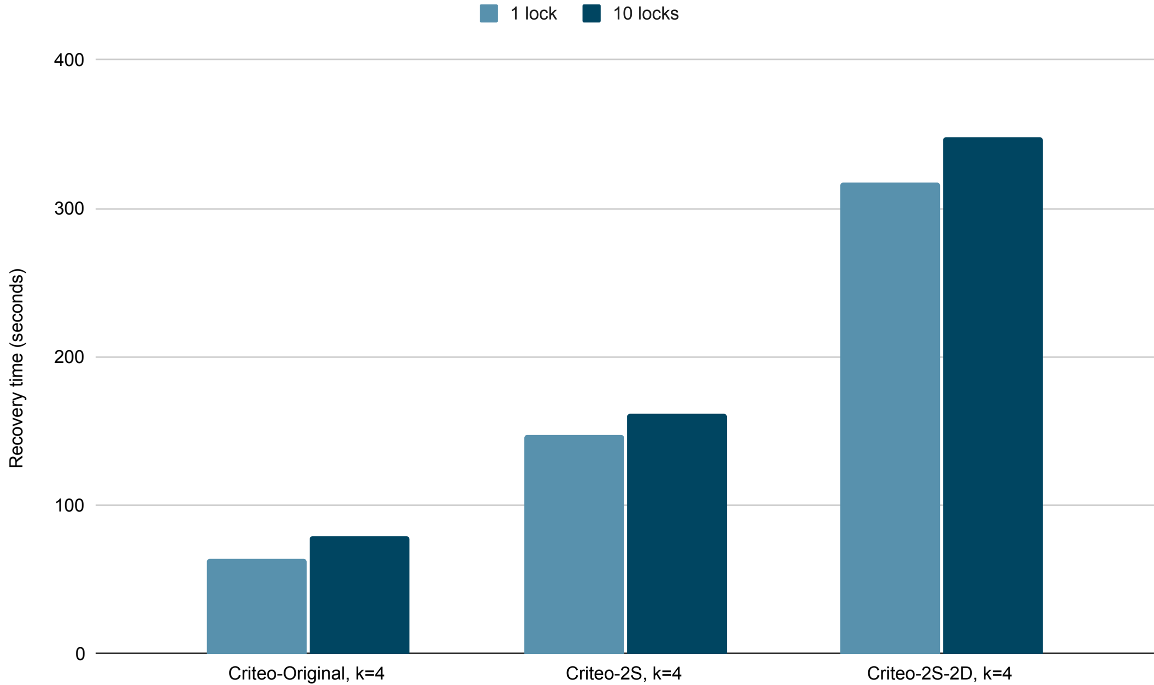

Figure 4.4: Effects of the number of partitions on recovery time

ery time increases more quickly with DLRM size than checkpointing due to the k-fold increase in data read and compute performed by a single server in ECRM when decoding. However, this does not significantly affect training in ECRM because ECRM can continue training during recovery with high throughput.

Effect of lock granularity We have discussed the idea of granular locks in §3.4.2. In order to evaluate the effect of locking granularity on recovery time, we compare the recovery time with a single lock with the recovery time using 10 partitions for each experimental setup. Figure 4.4 shows the effects of the number of partitions on recovery time. Using 10 granular locks with a 10% memory overhead increases the recovery time from 7.45% to 23.32%, mostly depending on the model size. The experimental results show that granular locking increases recovery time only by a moderate amount and demonstrates the applicability of granular locking. Meanwhile, the average training throughput during recovery remains the same level as with using a single lock.

4.3 Performance during normal operation

Figure 4.5 shows the training-time overhead of ECRM and checkpointing as compared to a system with no fault tolerance (and thus no overhead) in a four hour run. ECRM reduces training-time overhead in the absence of failure by 71.3%–88% and 41.3%–71.6% compared to Ckpt. 30 and Ckpt. 60, respectively. While the training-time overhead of checkpointing decreases with de-

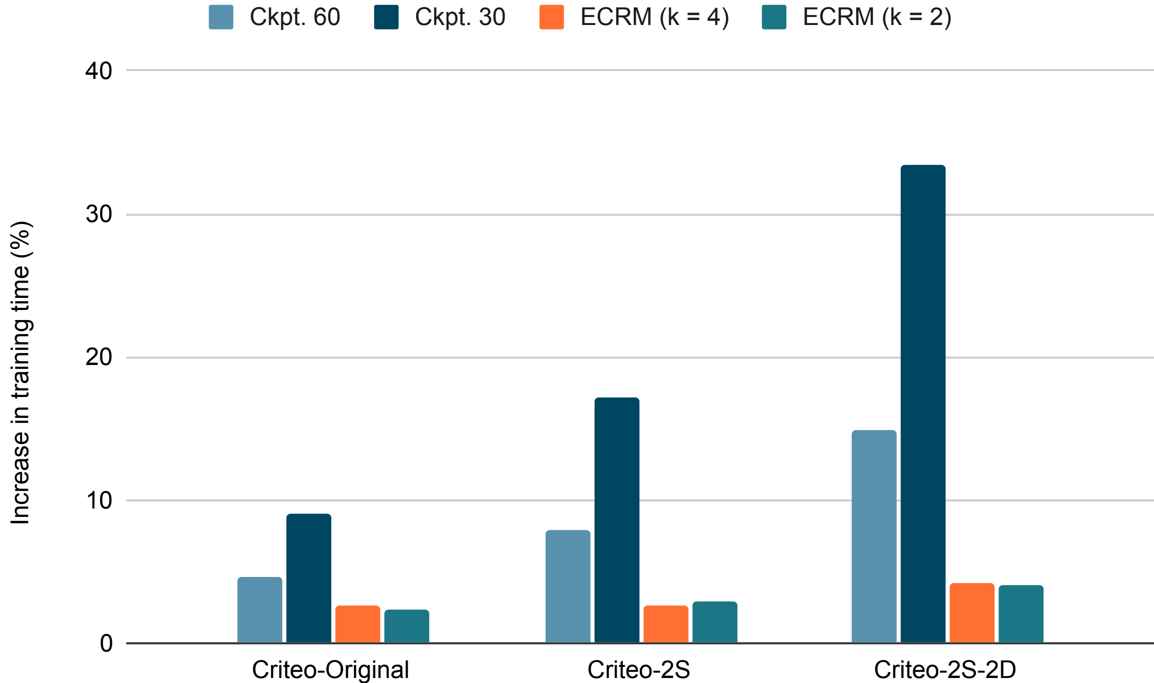

Figure 4.5: Training-time overhead in the absence of failures

creased checkpointing frequency, §4.2 showed that this came the expense of significantly worse recovery performance. Furthermore, ECRM's benefit over checkpointing grows with DLRM size. For example, on the 880 GB Criteo-2S-2D, Ckpt. 30 has training-time overhead of 33.4%, while ECRM has training-time overheads of 4.2% and 4% with k of 4 and 2, respectively. This illustrates the promise of ECRM for future DLRMs, which will likely grow in size [23, 31].

Training progress. Figure 4.6 plots the throughput of ECRM and Ckpt. 30 compared to training with no fault tolerance (No FT) on Criteo-2S-2D. As shown in the inset, ECRM has slightly lower throughput compared to No FT, while Ckpt. 30 causes throughput to fluctuate from that equal to No FT, to zero when writing a checkpoint. The effects of this fluctuation are shown in Figure 4.7: Ckpt. 30 progresses significantly slower than ECRM and No FT.

Effect of parameter k. As described in §3.6, ECRM has constant network bandwidth and CPU overhead during normal operation regardless of the value of parameter k. This is illustrated in Figures 4.5, 4.6, and 4.7, where ECRM has nearly equal performance with $k = 2$ and $k = 4$.

We also measure the training-time overhead of ECRM with $k = 10$ on a cluster twice the size as that described in §4.1 (to accommodate the higher value of k) and on a version of Criteo-Original scaled up to have the same number of embedding table entries per server as in the original cluster. In this setting, ECRM has training-time overhead of 0.5%. This smaller overhead stems not from the increase in parameter k, but from the decreased load on each server due to the increased number of servers. Nevertheless, this experiment illustrates that ECRM can support high values of k.

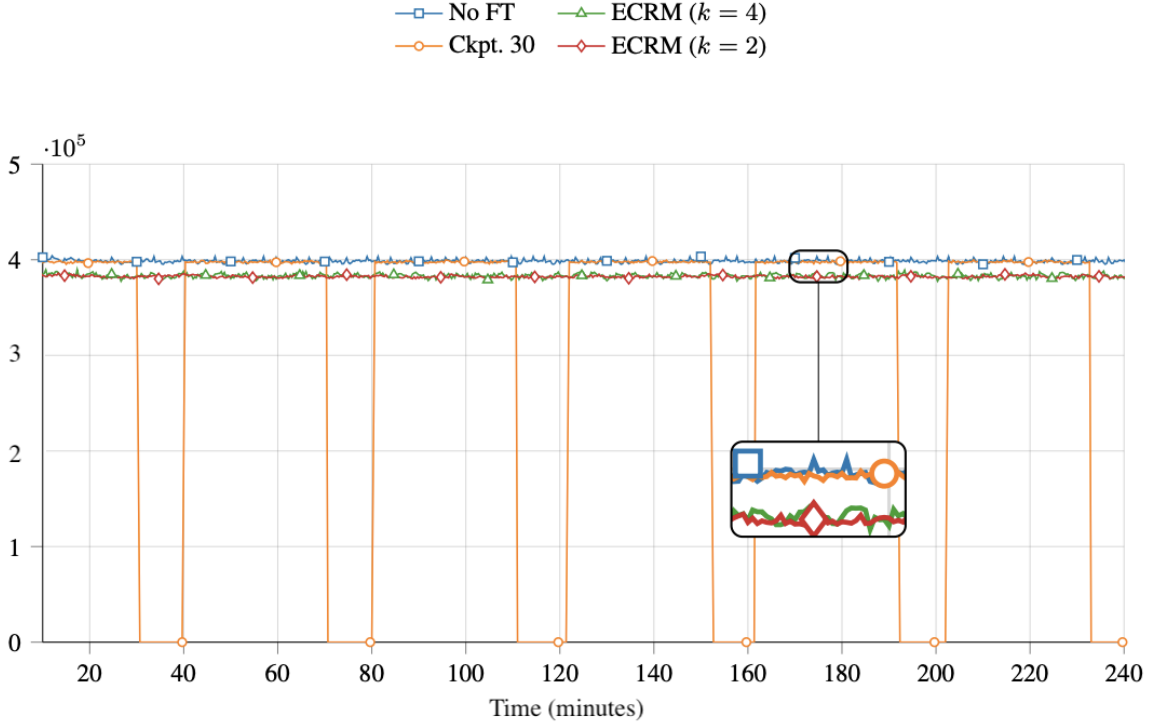

Figure 4.6: Throughput of training Criteo-2S-2D

Effect of ECRM on load imbalance. We next evaluate the effect of ECRM's approach to parity placement (§3.2) on cluster load imbalance. We measure the load imbalance by counting the number of updates that occur on each server when training Criteo-Original.

When training without erasure coding, the most-heavily loaded server performs 2.28× more updates than the least-heavily loaded server. In contrast, in ECRM with $k = 2$ and $k = 4$, this difference in load is 1.64× and 1.58×, respectively. This indicates that the increased load introduced by ECRM leads to *improved load balance.* Under ECRM, parities corresponding to the entries of a given server are distributed among all other servers. Thus, the same amount of load that an individual server experiences for non-parity updates will also be distributed among the other servers to update parities. While all servers will experience increased load, the most-loaded server is likely to experience the smallest increase in load because all other servers for whom it hosts parities have lower load. A similar argument holds for the least-loaded server experiencing the largest increase in load. Hence, the expected difference in load between the most- and least-loaded servers will decrease. Thus, while ECRM doubles the total number of updates in the system, its impact is alleviated by improved load balancing provided by its approach to parity placement.

Effect of a large number of workers To evaluate scenarios in which the servers in ECRM are more heavily-loaded, we additionally performed experiments with a different number of workers based on our current server setup. Figure 4.8 shows the average training throughput

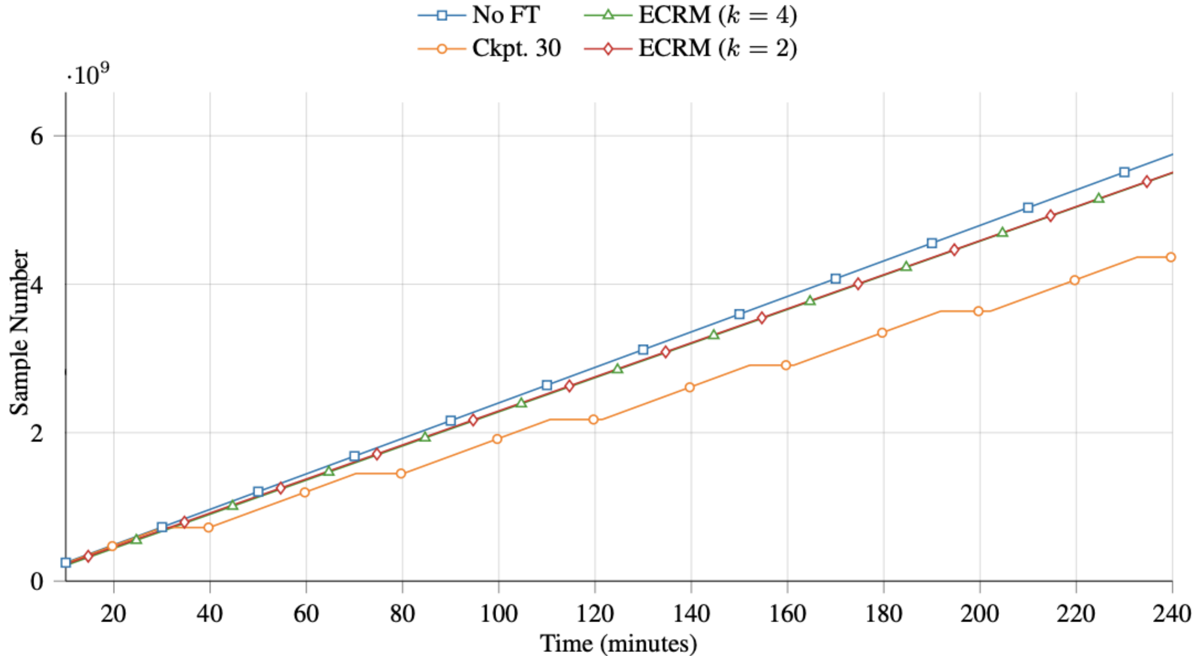

Figure 4.7: Progress of training Criteo-2S-2D

attained as the number of workers vary from 5 to 25, corresponding to $1\times$ to $5\times$ the number of the servers. As the number of servers increase, embedding table entries are accessed more frequently and therefore servers become more heavily loaded, which increases the severity of server-side bottlenecks. Compared to No FT approach, ECRM's overhead increases with the number of workers from 1.4% for 5 workers to 2.7% for 15 workers, and finally to 7.5% for 25 workers. Such increase is expected as ECRM adds load to servers in performing parity updates. Figure 4.8 also shows the training throughput of Ckpt. 30 with a constant overhead of 9.0%. The results show that even in settings with higher worker to server ratio, ECRM maintains lower overhead than checkpointing during normal operation.

Effect of reduced server computational and networking resources. As ECRM introduces CPU and network bandwidth overhead on servers during training, it is expected that ECRM will have higher training-time overhead when server CPU and network resources are limited. We evaluate ECRM in these settings by artificially limiting these resources when training Criteo-Original. To evaluate ECRM with limited server CPU resources, we replace the r5n.8xlarge server instances described in §4.1 with x1e.2xlarge instances, which have the same amount of memory, but $4\times$ less CPU cores. ECRM's training-time overhead with $k = 4$ is 11.1% when using these instances, higher than that on the more-capable servers (2.6%).

To evaluate ECRM with limited server network bandwidth, we replace the r5n.8xlarge instances (which have 25 Gbps) described in §4.1 with r5.8xlarge instances (which have 10 Gbps). ECRM's training-time overhead with $k = 4$ is 6.5% on these bandwidth-limited instances, higher than that on the more-capable servers (2.6%).

Even on these resource-limited servers, ECRM still benefits from significantly improved per-

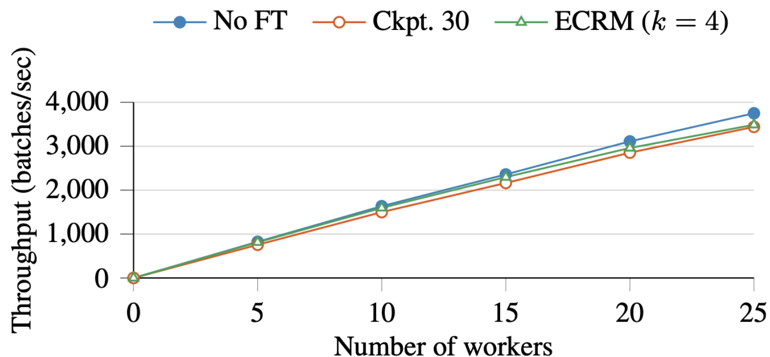

Figure 4.8: Average training throughput with varying number of workers during normal operation

formance during recovery compared to checkpointing and has training-time overhead comparable to Ckpt. 30 and slightly higher than Ckpt. 60.

Note that such limited resources represent a purposely unrealistic cases in the industry, due to the fact that custers in which production DLRMs are typically equipped with high-performance networks. A study by Facebook [6] reports that clusters used for DLRM training contain networks with 100 Gbps bandwidth and often utilize Infiniband to ensure network bandwidth is not the overall system's bottleneck.

Benefit of difference propagation. One of the motivations behind ECRM's approach of difference propagation described in §3.3.2 was to reduce the training-time overhead of keeping parities up-to-date. To illustrate this reduced overhead, we compare ECRM to the naive alternative, gradient propagation in training Criteo-Original. With $k = 4$, gradient propagation has a training-time overhead of 9.0%, while difference propagation has an overhead of only 2.6%. This illustrates the benefit of difference propagation in ECRM.

Chapter 5

Related Work

5.1 DLRM training and inference systems

Many aspects of DLRM systems have been explored: workload analysis [6, 15, 23], architectural support [19], system design [12, 17, 18], and model-system codesign [14]. To the best of our knowledge, ECRM is the first system to focus on efficient fault tolerance for DLRM training. Most the related work mentioned above are thoroughly discussed in the main thesis in §2.

5.2 Checkpointing

Checkpointing has long been a topic of intense study in high-performance computing in which the large-scale in which scientific simulations are performed requires efficient general-purpose approaches to fault tolerance [9, 26].

Recently, approaches to reduce the overhead of checkpointing in large-scale neural network training have begun to arise [28]. Some techniques take approximate checkpoints to reduce overhead [7, 30], but it is difficult for practitioners to reason about losses in accuracy due to such approximation. Other approaches continue training while writing a checkpoint [5], but this can result in inconsistent checkpoints; given the amount of time it takes to write checkpoints, many training updates may have been applied to the final model parameters being written since the time that the first parameters were written.

More closely related to our target setting of DLRM training, recent works have explored leveraging partial recovery [24] and checkpoint quantization [13] to reduce the overhead of checkpointing in DLRM training. However, like the approaches described above, these techniques can potentially change the trajectory of training by reloading approximate models after a failure has occurred. Our conversations with production DLRM training teams have indicated that such approximation is difficult for practitioners to reason about, and is thus avoided.

ECRM differs from the techniques above by (1) making use of erasure codes in novel ways to alleviate overheads associated with checkpointing, (2) specializing its design to the unique characteristics of DLRM training, and (3) introducing no additional inconsistency or accuracy loss to training.

5.3 Coding in machine learning systems

A line of work has explored the use of coding-theoretic ideas in machine learning systems. This work has primarily been applied to alleviating straggling workers in training limited classes of machine learning models (e.g., [11, 22, 25, 36, 38]) and serving neural networks [20, 21]. In contrast, ECRM imparts fault tolerance to DLRM training, which differs significantly in model architecture and system design to the settings considered by these works.

Chapter 6

Conclusion

ECRM is a new approach to fault tolerance in DLRM training that employs erasure coding to overcome the downsides of checkpointing-based fault tolerance. ECRM encodes the large embedding tables and optimizer state in DLRMs, maintains up-to-date parities with low overhead, and enables training to continue during recovery while maintaining consistency of recovered entries. Compared to checkpointing, ECRM reduces training-time overhead in the absence of failures by up to 88%, recovers from failures faster, and allows training to proceed without any pauses both during normal operation or recovery. While ECRM's benefits comes at the cost of additional memory requirements and load on the servers, the impact of these is alleviated by the fact that memory overhead is only fractional and that load gets evenly distributed. ECRM shows the potential of erasure coding as a superior alternative to checkpointing for fault tolerance in efficiently training current and future DLRMs.

Bibliography

[1] Display advertising challenge: Ctr terabyte ads data set. https://www.kaggle.com/c/criteo-display-ad-challenge. Last accessed 3 October 2020. 1

[2] MLPerf Training. https://mlperf.org/training-overview/, . Last accessed 10 September 2020. 1

[3] MLPerf Inference Github Repository. https://github.com/mlperf/inference, . Last accessed 10 October 2020. 4.1

[4] Introducing NVIDIA Merlin HugeCTR: A Training Framework Dedicated to Recommender Systems. https://tinyurl.com/yy82pd2l. Last accessed 10 September 2020. 1

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016. 5.2

[6] Bilge Alcun, Matthew Murphy, Xiaodong Wang, Jade Nie, and Kim Wu, Carole-Jean Hazelwood. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. *arXiv preprint arXiv:2011.05497*, 2020. 2.2.1, 4.3, 5.1

[7] Yu Chen, Zhenming Liu, Bin Ren, and Xin Jin. On Efficient Constructions of Checkpoints. In *Proceedings of the International Conference on Machine Learning (ICML 20)*, 2020. 5.2

[8] Paul Covington, Jay Adams, and Emre Sargin. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, 2016. 1, 2.1

[9] John T Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006. 2.2.2, 5.2

[10] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12(7), 2011. 2.1, 3.3.1

[11] Sanghamitra Dutta, Ziqian Bai, Haewon Jeong, Tze Meng Low, and Pulkit Grover. A Unified Coded Deep Neural Network Training Strategy Based on Generalized PolyDot Codes. In *2018 IEEE International Symposium on Information Theory (ISIT 18)*, 2018. 5.3

[12] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using Non-Volatile Memory for Storing Deep Learning Models. In *The Second Conference on Systems and Machine Learning (SysML 19)*, 2019. 1, 5.1

[13] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Murali Annavaram, Krishnakumar Nair, and Misha Smelyanskiy. Check-N-Run: A Checkpointing System for Training Recommendation Models. *arXiv preprint arXiv:2010.08679*, 2020. 5.2

[14] Antonio Ginart, Maxim Naumov, Dheevatsa Mudigere, Jiyan Yang, and James Zou. Mixed Dimension Embeddings with Application to Memory-Efficient Recommendation Systems. *arXiv preprint arXiv:1909.11810*, 2019. 5.1

[15] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The Architectural Implications of Facebook's DNN-based Personalized Recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA 20)*, 2020. 5.1

[16] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012. 1

[17] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, et al. XDL: An Industrial Deep Learning Framework for High-Dimensional Sparse Data. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, 2019. 1, 1, 1, 2.1, 2.2, 3.3.1, 2, 3.5, 4.1, 4.1, 5.1

[18] Dhiraj Kalamkar, Evangelos Georganas, Sudarshan Srinivasan, Jianping Chen, Mikhail Shiryaev, and Alexander Heinecke. Optimizing Deep Learning Recommender Systems' Training On CPU Cluster Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 20)*, 2020. 5.1

[19] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA 20)*, 2020. 5.1

[20] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity Models: Erasure-Coded Resilience for Prediction Serving Systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 19)*, 2019. 5.3

[21] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Learning-Based Coded Computation. *IEEE Journal on Selected Areas in Information Theory*, 2020. 5.3

[22] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. Speeding Up Distributed Machine Learning Using Codes. *IEEE Transactions on Information Theory*, July 2018. 5.3

[23] Michael Lui, Yavuz Yetim, Özgür Özkan, Zhuoran Zhao, Shin-Yeh Tsai, Carole-Jean Wu, and Mark Hempstead. Understanding Capacity-Driven Scale-Out Neural Recommendation Inference. *arXiv preprint arXiv:2011.02084*, 2020. 1, 2.2.2, 4.3, 5.1

[24] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark C Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. CPR: Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. *arXiv preprint arXiv:2011.02999*, 2020. 1, 2.2.1, 5.2

[25] Raj Kumar Maity, Ankit Singh Rawat, and Arya Mazumdar. Robust Gradient Descent via Moment Encoding with LDPC Codes. *arXiv preprint arXiv:1805.08327*, 2018. 5.3

[26] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10)*, 2010. 3.2, 5.2

[27] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *arXiv preprint arXiv:1906.00091*, 2019. 1, 2.1, 3.5, 4.1

[28] Bogdan Nicolae, Jiali Li, Justin Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *CCGrid'20: 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2020. 5.2

[29] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 88)*, 1988. 1, 2.3.2, 3.2

[30] Aurick Qiao, Bryon Aragam, Bingjing Zhang, and Eric Xing. Fault Tolerance in Iterative-Convergent Machine Learning. In *International Conference on Machine Learning*, pages 5220–5230, 2019. 5.2

[31] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory Optimization Towards Training a Trillion Parameter Models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 20)*, 2020. 1, 2.1, 2.2.2, 4.3

[32] K. V. Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *USENIX Workshop on Hot Topics in Storage and File Systems*, 2013. 1

[33] K. V. Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A Hitchhiker's Guide to Fast and Efficient Data Reconstruction in Erasure-Coded Data Centers. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM 14)*, 2014. 1, 1, 3.4.1

[34] Luigi Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols.

ACM SIGCOMM Computer Communication Review, 27(2):24–36, 1997. 1, 2.3.2

[35] Mahesh Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment*, 6(5), 2013. 1, 3.4.1

[36] Rashish Tandon, Qi Lei, Alexandros G Dimakis, and Nikos Karampatziakis. Gradient Coding: Avoiding Stragglers in Distributed Learning. In *International Conference on Machine Learning (ICML 17)*, 2017. 5.3

[37] Hakim Weatherspoon and John D Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, 2002. 1, 2.3.2

[38] Qian Yu, Netanel Raviv, Jinhyun So, and A Salman Avestimehr. Lagrange Coded Computing: Optimal Design for Resiliency, Security and Privacy. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS 19)*, 2019. 5.3