CHECKMATE: Zero Performance Overhead Model Checkpointing via Network Gradient Replication

Ankit Bhardwaj[*†], Weiyang Wang[*‡], Jeremy Carin[‡], Adam Belay[‡], and Manya Ghobadi[‡]

[†]Tufts University, [‡]Massachusetts Institute of Technology

Abstract

This paper presents Checkmate, a system that enables per-iteration checkpointing in DNN training without any training slowdown. The traditional approach to checkpointing requires a pause in training to copy model states to a separate location, allowing the state to be restored in the event of failure. This approach fundamentally has a tradeoff between the frequency of checkpoints and the cost of a failure. We avoid this tradeoff; our key insight is that in data-parallel training, all information necessary to create a checkpoint already exists in the network as gradients. Our core contribution is a new multicast abstraction that simultaneously delivers gradients to a separate CPU-based shadow cluster. The shadow maintains a checkpoint by applying those gradients to a copy of the model. Our evaluation shows that Checkmate performs per-iteration checkpointing with training throughput comparable to an ideal no-checkpoint baseline. Checkmate achieves 5 to 34.5× more frequent checkpointing compared to state-of-the-art checkpointing systems, resulting in 80% to 97.1% reduction in repeated work per failure. At the same checkpointing frequency, Checkmate delivers 1.3× to 6.5× throughput compared to other systems.

1 Introduction

Today's deep neural networks (DNNs) have hundreds of billions of parameters, and training them requires tens of thousands of GPUs for months at a time [3, 55]. At such a massive scale, failures are the common case. Meta's training of LLaMA3 encountered 419 failures [12], and Alibaba reported failure rates as high as 43% for its large training jobs [24].

The loss in progress caused by these failures is costly to operators. To help mitigate this cost, existing training frameworks periodically generate *checkpoints*, a process where they *interrupt* training to save model state to persistent storage [11, 38, 41, 56]. In the event of failure, training resumes from the most recent checkpoint, thereby minimizing the need to recompute work.

Unfortunately, existing checkpointing systems have high overheads, so they leave operators with a dilemma: frequent state saving reduces lost work but slows down training through its regular interruptions, whereas infrequent state saving reduces interruptions, but more work has to be recomputed in the event of a failure.

[*]Equal contribution.

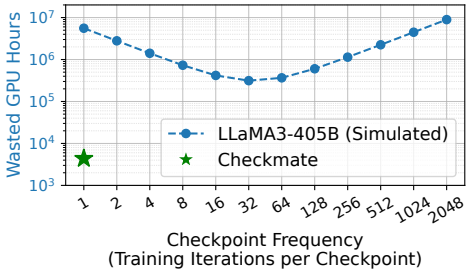


Figure 1: Total wasted GPU hours as checkpoint frequency changes, for LLaMA3-405B training. Existing checkpoint frameworks trade off normal-case efficiency with repeated work in failure cases, while Checkmate breaks this tradeoff.

Figure 1 presents a case study of this tradeoff using public data on the training of LLaMA3-405B. The total wasted GPU hours are plotted on the y-axis, defined as the sum of recomputation after failures and time spent on checkpointing, across different checkpoint frequencies (x-axis). On the right side of the plot, infrequent checkpointing wastes GPU hours due to recomputation on failure. For example, Meta's prior work [11] used a 30-minute checkpoint interval (between 256 and 512 iterations in Figure 1), resulting in 1.7 million wasted GPU hours, equating $15 million at current cloud prices [17]. On the other hand, checkpointing frequently wastes GPU hours due to the time it stalls training. For instance, the left-most point of Figure 1 represents checkpointing every iteration. It minimizes repeated work but incurs high overall waste, as each checkpoint slows the normal iteration time by 28%. Even at the best checkpoint frequency (every 32 iterations), the system still wastes over 300,000 GPU hours, costing $3.3 million.[1] Existing checkpointing techniques [37, 38, 40, 54, 56] reduce overheads but cannot break this fundamental tradeoff (detailed analysis in §6.7).

In this paper, we introduce Checkmate, a network-assisted checkpointing system that resolves this dilemma, enabling checkpointing every iteration without slowing down training. This is achieved through two key observations. First, model updates are deterministic: given a model state at time t and its corresponding gradients, applying the optimizer step produces the state at time $t+1$, whether for training *or* checkpointing. Second, in data-parallel training, these gradients are already computed and exchanged over the network every iteration. Checkmate captures these in-flight gradients and applies them to a prior checkpoint replica on a separate CPU cluster, re-

[1]Our estimates are based on Meta's LLaMA3 technical report and public cloud GPU pricing with detailed derivations in Appendix A and B.

constructing the latest model state without GPU involvement. Together, these insights enable per-iteration checkpointing with virtually no training overhead.

To realize this design, Checkmate introduces a *multicast-update* framework that offloads checkpointing to the network and a dedicated cluster of CPU nodes. It replicates reduced gradients directly at programmable switches and *multicasts* them to shadow nodes, which apply these *updates* to independently maintained model replicas. The checkpointing process remains transparent to GPUs, allowing training to proceed without disruption. This allows Checkmate to substitute wasted GPU hours on recomputation with cheaper CPU hours on checkpointing. For the aforementioned LLaMA3-405B model, the shadow cluster consumes only 166,000 CPU-node hours to enable per-iteration checkpointing, reducing wasted GPU time from over 300,000 hours to just 4,367 as shown in Figure 1, cutting GPU waste by over 98% and saving an estimated $2.6 million in training costs.

Checkmate must meet three key requirements to function correctly and at scale: *selecting* the gradients to replicate from the network, *reliability* in delivering them, and *timeliness* in applying updates fast enough to keep up with training.

For *selection*, Checkmate faces the challenge that the network carries many types of traffic during each iteration. Checkmate must select the final version of the gradients exactly once per iteration. We develop a lightweight tagging mechanism that marks final gradients precisely once per iteration for replication (§4.1). To ensure all tagged traffic reaches the shadow cluster, our system strategically places switches and shadow nodes in the network topology, guaranteeing all tagged gradients reach their multicast destination (§4.4). This requires additional switch ports for connecting shadow nodes, a necessary overhead to ensure scalability.

For *reliability*, Checkmate must ensure all tagged gradients reach the shadow cluster without packet loss or corruption, since losing gradient updates would result in invalid checkpoints. It avoids congestion by using dedicated switch ports for shadow nodes, thereby isolating replication traffic from the training traffic. Checkmate enables Priority Flow Control (PFC), a widely adopted mechanism in training networks to handle transient receiver-side pressure. (§4.3)

For *timeliness*, Checkmate must ensure that shadow nodes, despite being CPU-based, can keep pace with the GPU training rate. Checkmate exploits the fact that each parameter's update is independent for popular optimizers and deterministically partitions the model state across the shadow cluster (§4.2). Scaling out the shadow cluster also ensures that flow control mechanisms are rarely triggered.

We build a small-scale prototype of Checkmate and evaluate it using popular vision and language DNN models on a 16-node testbed, comprising 12 training nodes equipped with NVIDIA A100 80 GB GPUs and 4 CPU-based shadow nodes, connected by 100 Gbps network interface cards (NICs). Checkmate achieves per-iteration checkpointing with virtually no impact on training throughput, matching the no-checkpoint performance across all models. Compared to production systems like PyTorch [41] and state-of-the-art research prototypes like CheckFreq [38] and Gemini [56], Checkmate achieves 5 to 34.5× more frequent checkpointing, resulting in 80% to 97.1% reduction in repeated work per failure. At the same checkpointing frequency, Checkmate delivers 1.3× to 6.5× throughput (§6.2). Our simulation shows that at 16K GPU scale, Checkmate saves 70,000 GPU hours over a 54-day run, even with a failure rate at 0.5% of Meta's reported value (§6.7). The Checkmate codebase is open source [7].

2 Background

2.1 Overview of Distributed DNN Training

Training large models requires splitting computations across multiple GPUs to manage large datasets. In data parallel (DP) training, each GPU holds a full copy of the model and processes different batches of training data in parallel. When models become too large to fit on a single GPU, pipeline parallelism (PP) is employed to divide the model into stages on different GPUs [39]. In practice, large-scale training often combines DP and PP to scale both models and datasets, where each pipeline stage is independently replicated across a group of GPUs as a DP group.

Each training iteration involves three phases: forward, backward, and optimizer step. In the forward pass, each DP group processes a batch of data. The backward pass computes gradients and synchronizes them with other DP groups. Finally, the optimizer uses these gradients to update model parameters. Frameworks like PyTorch divide gradients into fixed-size buckets to facilitate more efficient communication.

Gradient synchronization algo. GPUs perform gradient synchronization using an operation called *AllReduce*. AllReduce typically consists of two stages: *ReduceScatter* followed by *AllGather*. During ReduceScatter, each node splits its gradients into chunks, exchanges these chunks with other nodes, and reduces them (e.g., by summing or averaging). After this phase, each node holds a different chunk of the fully reduced result. In the following AllGather phase, each node sends its reduced chunk to another node. This process is repeated multiple times until every node has all the reduced chunks.

2.2 Training Failures

As model sizes increase and training scales grow, failures during the training process become more common. On each failure, the entire training job must restart. To avoid losing all training progress, practitioners periodically save the model state to a reliable medium, a process known as *checkpointing*.

Existing checkpointing approaches use *copy-persist* mechanism: they first copy model states (i.e., model parameters and optimizer states) out of GPU memory and then persist them to a reliable medium. The most straightforward approach is

to pause training and perform checkpointing *synchronously*, where each GPU waits for the copy-persist operation to complete before resuming training, resulting in longer GPU stalls.

Recent work, such as CheckFreq [38], DataStates-LLM [37], Gemini [56], and PyTorch Distributed Checkpointing [41,42], attempts to mitigate these stalls by making *copy-persist asynchronous*. These systems overlap the copy step with forward and backward passes, as the model state is quiescent during this time. The persist step is then completed asynchronously before the start of the next checkpoint operation, reducing the overall stall time. PyTorch Distributed makes further improvements by sharding checkpoints across training nodes, while CheckFreq dynamically tunes the checkpointing frequency to balance overheads and lost work on recovery. DataStates-LLM [37] lazily streams and persists LLM artifacts to the background. To overcome limited disk bandwidth and high I/O latencies, Gemini [56] saves checkpoints to remote memory within the training cluster, using spare training network bandwidth to copy model states.

Although state-of-the-art approaches reduce training stalls, this paper shows that these schemes have fundamental overheads. Specifically, we identify two primary overheads. First, the copy operation requires GPUs to clone states in parallel with training, which interferes with the training computations. Second, the persist operation must finish before the start of the next checkpoint to prevent unbounded memory usage, which ultimately limits the checkpointing frequency.

3 Motivation

3.1 Need for Per-Iteration Checkpointing

While checkpointing prevents a complete restart, progress loss is still inevitable, as any training progress made after the last checkpoint is lost and must be repeated on recovery. As models and cluster sizes grow, this repeated work becomes increasingly costly. For example, during LLaMA3-405B training on 16K GPUs, a single failure requires repeating 4,096 GPU-hours of computation on average when checkpointing every 30 minutes. Throughout the entire training, Meta reported 419 failures and wasted compute adds up to as much as $15 million at current Google Cloud prices [17].

With *per-iteration* checkpointing, failure recovery incurs the absolute minimum repeated work. Checkpointing every iteration saves millions of dollars for a single job in large-scale training by reducing repeated work. However, existing systems introduce substantial checkpointing overhead, slowing down training and consuming additional resources during the checkpoint process itself. The following subsection quantifies the overheads in today's state-of-the-art systems.

3.2 Limitations of Existing Solutions

Existing systems reduce checkpointing overhead but cannot eliminate it, especially at per-iteration frequency. The fundamental limitation lies in the *copy-persist* model: copying from GPU to CPU memory introduces stalls due to the bandwidth gap between GPU and CPU, while persisting data to storage media adds further delays. With fast training iterations, these combined latencies prevent checkpointing from fully overlapping with computation, leading to unavoidable GPU stalls.

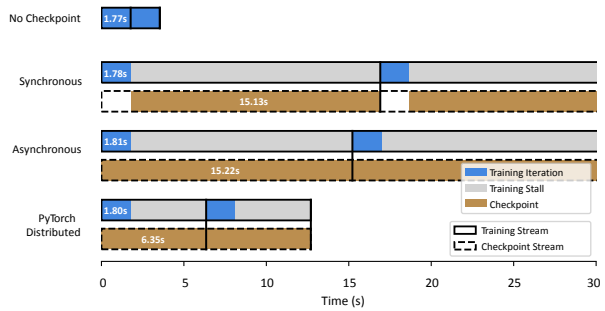


Figure 2: GPT3-XL training iteration times and stalls with various checkpointing approaches. Existing approaches do not fully remove stalls when checkpointing per-iteration.

To quantify the overheads of existing systems, we compare four checkpointing techniques to illustrate this limitation. We measure the iteration time and checkpoint stalls for GPT3-XL with a batch size of 16 on a 4-node setup (details in §6.1). Figure 2 highlights these measurements for two iterations, for comparison. The average iteration time for this setup is 1.77 seconds (as shown in the top-most row in Figure 2). The simplest strategy, synchronous checkpointing, performs the worst, causing a 9.5× slowdown by stalling training during the entire checkpoint. Asynchronous checkpointing hides some of this time by overlaying the start of the checkpoint with training, but ultimately must still copy and persist the same total volume of data, causing a slowdown of 8.45×. PyTorch Distributed reduces the data volume each node needs to persist by sharding the checkpoint. Even still, we observe a 3.5× slowdown when sharded across four nodes. While smaller shards on a large cluster would reduce these slowdowns, our experimental results show that even with smaller shards, the overheads are still significant. Gemini aims to achieve per-iteration checkpointing, but our evaluations show that the training throughput slows by up to 3.47× when checkpointing every iteration (§6.2).

When checkpointing per-iteration, current systems slow training across most model sizes. While further optimization of copy-persist is possible, we propose rethinking the problem by *removing the GPU from the checkpointing process*.

3.3 Pauseless Checkpointing Per-Iteration

To enable checkpointing every iteration without disrupting training, we seek to remove the GPU from involvement in checkpointing entirely. At first glance, this goal appears impossible, as the model state lives in GPU memory. However, this implication is only valid if one focuses solely on extracting

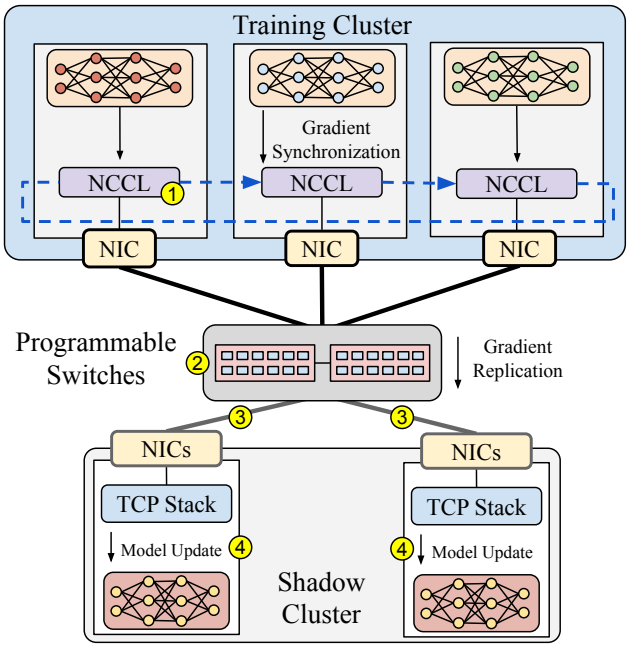


Figure 3: Overview of Checkmate's architecture. The diagram highlights the interaction between training nodes, the network switch, and shadow nodes to enable pauseless checkpointing.

checkpoint data directly from the GPU. Instead, we ask a different question: what data is already available during training that we can reuse for checkpointing rather than copying the entire model each time?

Our first insight is to *reuse existing checkpoints*. Model updates are deterministic functions of the previous model states and the *gradients* during training. Hence, we can apply the same optimizer step if we have the gradients and a copy of the prior model *as a checkpoint*. This insight opens up *incremental* update for per-iteration checkpointing.

Our second insight is that these gradients are *already in the network*. In DP training, GPUs exchange gradients every iteration to synchronize updates. Hence, rather than extracting data from the GPU, we tap into this existing data flow and replicate the gradients to a dedicated checkpoint cluster.

Checkmate combines these two ideas and introduces a *multicast-update* framework for checkpointing. In Checkmate, we replicate the gradients *in-network* using hardware multicast and stream them to a pool of CPU "shadow" nodes. These nodes apply the same updates as the GPUs to maintain consistent model copies, enabling per-iteration checkpointing with no involvement from the training GPUs.

4 Checkmate System Design

Overview: Checkmate enables pauseless per-iteration checkpointing. It *selects* the gradients to replicate from the network, delivers them *reliably* to the shadow nodes, and applies updates fast enough to keep up with training in a *timely* manner. It achieves this through a three-way collaboration across the training nodes, the network switches, and dedicated shadow CPU nodes. Figure 3 highlights Checkmate's main steps:

① **Tagging gradients on training nodes.** Checkmate tags gradients before transmitting over the network, enabling the network data plane to distinguish them from other traffic.

② **Multicasting in the switch.** The switch multicasts tagged gradients to their training destination and the appropriate shadow nodes, while forwarding other packets normally. Priority Flow Control (PFC) ensures lossless delivery.

③ **Receiving gradient on shadow nodes.** Shadow nodes receive the gradients and map them to the correct model layers.

④ **Running optimizer step on shadow nodes.** Once all gradients for an iteration are received, shadow nodes run the optimizer step, updating their model and optimizer state.

Using these steps, Checkmate ensures that shadow nodes remain up to date with the training state, enabling failure recovery at a per-iteration granularity.

4.1 Gradient Tagging on Training Nodes

To ensure correct and reliable operation, Checkmate must deliver each reduced gradient to the shadow cluster exactly once per iteration. Any duplication or loss would result in an inconsistent checkpoint, ultimately leading to a failed recovery. This requirement is at odds with the behavior of standard collective algorithms, such as Ring AllReduce, where training nodes exchange the same gradient chunks multiple times over several communication rounds. During the AllGather phase in Ring AllReduce, each node sends and receives a portion of the reduced gradients in each round, repeatedly transmitting the same data over the network. These algorithms improve bandwidth utilization but complicate our attempt to extract a single, complete copy of the gradients from the network.

To accurately identify reduced gradients in the network, Checkmate tags them on the training nodes before transmitting. While our current design is based on NCCL's most widely used Ring AllReduce algorithm, this tagging approach is generalizable to other AllReduce algorithms as well (§8).

The AllGather phase in Ring AllReduce exchanges final gradients $(n-1)$ times over the network in a single round for n training nodes. A naive approach is to tag them in the first round. However, this leads to an n-way incast problem on the shadow node, where all n training nodes simultaneously transmit their gradients to one shadow node, resulting in network congestion and data loss on shadow nodes. Instead, Checkmate uses a heartbeat algorithm, where training nodes tag their gradient chunks evenly across $(n-1)$ AllGather rounds, distributing transmissions more uniformly.

4.1.1 Heartbeat-based Tagging

The heartbeat-based tagging mechanism is applied during $(n-1)$ AllGather rounds. In a ring of n ranks, each rank sends one chunk of reduced gradients to its next neighbor and receives another chunk from its previous neighbor for

Chunk-0	Chunk-1	Chunk-2	Chunk-3

(a) Snapshot of the bucket for Rank-0 during round 0.

GPU	Round 0	Round 1	Round 2
0	**C1**	C0	C3
1	C2	C1	C0
2	C3	C2	C1
3	**C0**	**C3**	**C2**

(b) C1 and C0 are tagged in the first round, and C3 and C2 in the remaining two rounds. Each chunk is tagged exactly once.

Figure 4: AllGather tagging example for a 4-GPU cluster.

$n-1$ rounds. For example, Rank-0 starts iteration 0 holding Chunk 1 (see figure 4a); it forwards that chunk to Rank-1 and simultaneously receives Chunk 0 from Rank $n-1$. After $n-1$ rounds every rank has gathered all n chunks. Checkmate minimally modifies the algorithm to tag chunks. To ensure that each chunk is tagged exactly once, the algorithm only tags them on the boundary ranks: Rank-0 tags only its first chunk, and the last rank tags all its chunks before sending. This approach ensures full coverage with minimal overhead.

Figure 4b illustrates how the algorithm distributes tagging across AllGather rounds on a 4-node cluster while also ensuring that all gradients are tagged precisely once. Rank-0 tags only in the first round and Rank-3 in all the rounds, tagging C1, C0, C3, and C2, respectively. This example shows a simplified version of the actual implementation. In practice, Checkmate handles multiple channels and connections, which are dynamically tuned by NCCL at runtime. Importantly, Checkmate does not alter any other aspects of the algorithm or runtime, apart from introducing a lightweight 1-bit tagging step that has no measurable impact on performance.

While this approach reduces the likelihood of network incast on the shadow nodes, it does not eliminate it. In AllGather, n gradient chunks are transmitted in only $(n-1)$ rounds. Consequently, one round inevitably involves two ranks transmitting data to a shadow node simultaneously (C0 and C1 in the first round in Figure 4b example). To handle the bandwidth requirement, Checkmate equips each shadow node with two NICs, doubling their network bandwidth compared to training nodes. This setup allows shadow nodes to manage parallel transmissions from two training nodes without loss or congestion, ensuring *reliable* reception of all gradients. The dual-NIC approach does not significantly increase costs, as Checkmate requires only a few shadow nodes to keep up with training. We later show that only one shadow node is enough to handle large-scale models (e.g., GPT3-6.7B) (§6.3).

4.1.2 Additional Metadata for Reassembling Buckets

The heartbeat tagging algorithm ensures that the network receives a full copy of the tagged gradient. However, from the network flow's perspective, the algorithm tags non-consecutive data chunks in a continuous stream. This, coupled with parallel transmissions over multiple channels, makes gradient reassembly on the shadow nodes challenging. To address this, the network layer maintains a separate sequence number for each channel, incrementing it only for tagged chunks. Then, it inserts the sequence number as a custom TCP option into the packets before sending them to the network. The network switch mirrors each tagged packet and substitutes its original TCP sequence number with the channel-specific counter. This allows the shadow node to view the mirrored traffic on each channel as a single, continuous TCP stream.

Next, we focus on how shadow nodes maintain a checkpoint.

4.2 Shadow Cluster

In this section, we describe the operations of the shadow cluster. Each node in the cluster performs three tasks to maintain the checkpoint: (1) it receives mirrored packets from the network and reassembles them into buckets, (2) it then maps each bucket to its corresponding layer in the model, and (3) it runs the optimizer step to maintain the checkpoint.

4.2.1 Reassembling Buckets on Shadow Nodes

Checkmate exploits the deterministic ordering guarantees of collective communication libraries (e.g., NCCL) to infer exactly which data chunks and offsets arrive on each channel. This same determinism, which enables GPUs to run AllReduce and other collectives without branching, allows for predicting the sequence of incoming data for each channel on the shadow nodes. Checkmate then uses this information to set up channels similar to those on the receive side of the training nodes, calculating the expected chunk sizes and offsets for each channel to sequence the incoming data and reassemble it into the original bucket.

Each shadow node allocates two sets of channels: one set to handle chunks from the first rank and another for the last, as tags are added only on these boundary ranks. It then binds these channels to the two NICs in a round-robin manner to take full advantage of both NICs. During the first AllGather round, when both first and last ranks are tagging data, both NICs operate at full capacity. In subsequent rounds, each NIC operates at half capacity, avoiding scenarios where one NIC is idle while the other is near saturation.

4.2.2 Mapping Bucket to Model Parameters

After reassembling the buckets, each node must map these buckets to their corresponding layers in the model. While this process can be complex, Checkmate handles the mappings in a way similar to the framework used on the training nodes. Training frameworks like PyTorch group parameters by bin-packing them, starting from the last model layer and working backwards to the first. A model layer is mapped to a bucket until the bucket size is less than the maximum given size, such as 25MB in PyTorch DDP [32]. If a layer size exceeds the bucket size, it is mapped to a single dedicated bucket. While Checkmate can maintain a different mapping to optimize the

Listing 1 Sample code for distributed training in PyTorch

```
1  model = DistributedDataParallel(dnn)
2  optimizer = Optimizer(model.parameters())
3  for batch, expected in zip(inputs, expected_outputs):
4      optimizer.zero_grad()
5      output = model(batch)
6      loss = loss_fn(output, expected)
7      # Gradient computation and synchronization
8      loss.backward()
9      optimizer.step()
```

gradient capture from the network. The current design choice ensures that Checkmate easily integrates with existing training pipelines without additional overhead. Besides the ease of integration, the approach also avoids the need for additional memory allocation. During the optimizer step, each model layer points to a specific offset in a bucket for the gradients without making additional copies.

4.2.3 Updating the Checkpoint

Once all the buckets are mapped, each shadow node runs the optimizer step to update the checkpoint (the parameter and optimizer states). Each node runs the same loop as the training nodes, except shadow nodes skip the forward and backward passes and are replaced with the gradient receive logic from the switch.

Listing 1 shows the code running on the training nodes. The code runs a forward pass (line 5), calculates the loss (line 6), and runs a backward pass for each batch (line 8). Internally, the backward pass overlaps gradient computation and synchronization across different layers in the model. Lastly, the loop runs the optimizer step to update the model parameters and optimizer state (line 9).

Listing 2 shows the code running on the shadow nodes. It skips forward and backward passes altogether. Instead, shadow nodes wait for the gradients to arrive from the switch (line 6) and run the optimizer step on CPUs to update the model states (line 7).

Although the optimizer update is computationally cheaper compared to forward and backward passes, a single shadow node can still become a bottleneck for large models. To address this, Checkmate allows scaling the optimizer step across multiple shadow nodes.

4.2.4 Scaling out the Optimizer Step

Checkmate allows scaling the optimizer step across multiple nodes, ensuring shadow nodes keep pace with training nodes, even for large models. We make no assumptions about the compute capabilities of shadow nodes, and allow users to configure the number of nodes based on model requirements. Before starting training, Checkmate profiles shadow nodes and configures the system for optimal performance.

For seamless scaling, Checkmate opts to support *functional* optimizers, where the optimizer step for each parameter is deterministic and independent of the others. Most optimizers, including SGD [47], Adam [29], and AdamW [35], are

Listing 2 Sample code for shadow nodes

```
1  model = DistributedDataParallel(dnn)
2  optimizer = Optimizer(model.parameters())
3  buckets = model.get_buckets()
4  while True:
5      optimizer.zero_grad()
6      buckets.recv()
7      optimizer.step()
```

functional. This property enables Checkmate to distribute the optimizer step across multiple nodes without affecting algorithmic correctness or introducing synchronization overhead.

Finally, distributing buckets across multiple nodes requires the switch to identify the correct shadow node for delivery. Checkmate encodes the shadow node ID in each packet for the switch to replicate gradients appropriately.

Consolidating Checkpoint State for Recovery: Distributing the optimizer step adds an extra step in the recovery process. Checkmate uses a configurable timeout to consolidate shards into a complete checkpoint. After consolidation, each shadow node serves as a checkpoint to the training nodes simultaneously, enabling a quick recovery process.

4.3 Network Gradient Multicasting

This section describes how the programmable switches, located between the training and checkpoint nodes, intercept and mirror packets tagged by the training nodes, and reliably deliver them to the checkpoint nodes.

4.3.1 Switch Control-plane Setup

The switch control plane must be configured before it can deliver gradients to the shadow nodes. It involves three steps:

First, the control plane is configured with the network addresses for the boundary ranks for each DP group. The switch uses this metadata to distinguish gradients from different shards. For example, in a two-shard pipeline, both shards tag their gradients at the boundary ranks, and the switch uses those tags, along with the rank addresses, to tell them apart. For each DP shard, the control plane creates *protocol-independent multicast groups* for packet replication. For the first and last rank in a shard, the control plane identifies corresponding shadow nodes. It creates a *multicast group* with the next rank (training) in the DP shard and the shadow nodes. These multicast groups are inserted into a match-action table, later used for multicasting tagged gradient packets.

Second, the switch requires a mapping between the shadow node ID, used for optimizer scale out, and shadow node IP addresses. The training nodes set the shadow node IDs to scale up the optimizer step, and the switch uses these mappings to update the destination on mirrored packets.

Lastly, the switch implements a minimal TCP server to emulate a parameter server for shadow nodes. So, before starting the training, each shadow node establishes TCP connections with the switch, which are later used to forward gradients over TCP streams. It drops ACK packets from shadow nodes.

4.3.2 Intercepting and Multicasting Gradients

The data plane handles gradient multicasting statelessly. The switch performs regular L2 forwarding for untagged packets and match-action logic to process tagged ones. By carrying all necessary metadata within the packets, the switch avoids maintaining state, reduces complexity, and ensures adaptability to less complex switch architectures.

The ingress pipeline assigns a multicast group to the tagged packets based on control plane configurations and performs standard L2 forwarding for untagged ones. At the end of the ingress pipeline, packets destined for shadow nodes undergo additional processing. The switch replicates the packets according to their multicast group. All packets are then handed off to the corresponding egress pipeline for transmission.

The egress pipeline rewrites the sequence number to the sequence number expected by the shadow nodes, stored in the TCP option field by the training nodes (§4.1.2). It also updates the source and destination IPs for the shadow node's TCP stream. Finally, the packet is forwarded to its destination.

4.3.3 Lossless Packet Delivery

Checkmate requires all tagged gradients to reach their destination for a correct and consistent checkpoint. This requirement necessitates a lossless network for packet delivery. Because training traffic saturates links only up to line rate and multicast mirrors the same data rate, Checkmate avoids introducing additional congestion by design.

Packet loss can still occur at the shared packet buffers within the switch or due to transient slowdowns in packet consumption at the shadow nodes. To avoid this issue, the system utilizes Priority Flow Control (PFC) for connections with shadow nodes. In networks like RoCE, PFC is typically enabled system-wide, so no extra tuning is needed. In conventional datacenter networks, efficiently using PFC can be challenging [21], but recent studies show that PFC works well for predictable patterns in large-scale training [12].

Checkmate leverages protocol-independent multicast groups to facilitate efficient lossless delivery. This feature enables configuring multicast with arbitrary fields in the packet header. Furthermore, modern switches enable all downstream devices in a multicast group to apply backpressure to the upstream source during congestion, effectively managing network bottlenecks and preventing packet loss for tagged traffic. We tune PFC parameters, such as pause thresholds and buffer sizes, to align with workload requirements in both the training and the shadow cluster. This tuning maintains a lossless network, even under high traffic loads, ensuring reliable delivery of gradients for checkpoint correctness.

4.4 Network Resource Planning

This section analyzes Checkmate's network resource requirements and explains the topology choices to ensure scalability and performance isolation.

Network resource requirement. To replicate gradients in each iteration, Checkmate employs two multicast streams per DP group, regardless of the group's size. As a result, the total resource requirement scales linearly with the number of DP groups. For instance, the primary phase of Meta's LLaMA3-405B training utilized 128 DP groups. In this configuration, replicating gradients needs 256 multicast streams. Hence, Checkmate requires 256 additional switch ports and line-rate NICs, as well as 512 corresponding transceivers to support replication. Using cost numbers reported in prior work and market prices [13, 55], this amounts to $837,376. For a 16K-GPU setup, this amounts to less than 0.8% more network resource of the training cluster, a modest overhead to ensure the reliability of replication traffic, and isolating them from the training path. Furthermore, multicast is a hardware-supported primitive on modern datacenter switches, enabling gradient replication at line rate.

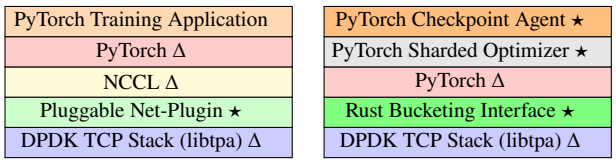
In addition to network resources, Checkmate requires a dedicated shadow cluster to update checkpointing. In the prior example, the shadow cluster needs to support 256 line-rate NICs and the corresponding CPU resources to update 128 shards of the model (at three billion parameters per shard). This setup requires at most 128 shadow nodes, each node costing $6,738 [50]. We assume that each node has a 28-core Intel Xeon Gold 5420+ Processor and 8×32 GB DRAM, similar to our testbed. Hence, the entire shadow cluster's compute resource costs $862,464. The total cost of compute and network resources required for the shadow cluster is equivalent to the price of seven DGX H100 GPU servers, at 0.31% of the 16K-GPU training cluster [49]. We quantify the operational cost and benefit of Checkmate in Section 6.7.

Where to put the shadow nodes. Checkmate must observe gradient synchronization traffic for all DP groups to facilitate replication. In hybrid-parallel training, DP synchronization traffic typically traverses the network even when faster interconnects like NVLink are present [12, 55]. Crucially, all inter-node traffic must pass through the Top-of-Rack (ToR) switches by definition, making ToR the ideal point for capturing and replicating DP traffic. Checkmate connects shadow nodes to the ToR layer, enabling multicast-capable switches to observe and replicate gradient traffic directly.

How to add additional ports. There are several ways to accommodate this requirement: one option is to deploy leaf switches with a higher port density to absorb the additional load. Alternatively, some existing uplink ports can be repurposed to connect shadow nodes, slightly increasing the oversubscription ratio, a tradeoff made acceptable by recent work showing that large LLMs can be trained effectively even on spine-free topologies [55].

5 Implementation

We implement a prototype of Checkmate, comprising ≈ 10000 lines of Rust, C, C++, Python, and P4 code for various com-



(a) Training. (b) Shadow cluster.

Figure 5: Software-stack running on training and Shadow nodes. ⋆ indicates components we implement, while Δ indicates components we apply minor changes only.

ponents on the training nodes, shadow nodes, and the programmable switch. We describe the implementation of each component below.

Training nodes: Figure 5a shows the software-stack on training nodes. Each node runs an unmodified PyTorch application while PyTorch itself has 50 lines of code changes, mainly to expose the bucketing interface to the user. Checkmate uses NCCL v2.22.3 for collective communication, with only ten lines of code changes to tag the reduced gradients before sending them to the network stack.

The majority of our implementation efforts were focused on developing a NCCL network plugin (≈ 5000 lines of Rust code). NCCL allows using a custom network stack at runtime based on `NCCL_NET_PLUGIN` environment variable. If the variable is set, NCCL loads the plugin and uses it for communication. This plugin is used to send and receive messages between the nodes [8]. The plugin handles NCCL API calls and adds tags and sequence numbers to the reduced gradients before sending them to the underlying TCP library.

The plugin uses libtpa, a DPDK-based TCP stack [36], for network transport. We modify libtpa to support custom sequence numbers and packet tagging (with a DSCP bit). Our plugin achieves up to 98.4 Gbps bus bandwidth in NCCL AllReduce on a 100 Gbps NIC, matching the performance of NCCL's InfiniBand implementation. The plugin's modular design also allows using it with other NCCL-based libraries.

Shadow nodes: Figure 5b shows the software-stack on Shadow nodes. We implement the checkpointing agent using the same high-level PyTorch code as the training nodes, but without forward and backward passes. Instead, it receives gradients from our custom bucketing interface to retrieve and store them. The bucketing interface internally uses the DPDK-based TCP stack to communicate with the switch. The network stack is fast and saturates two 100 Gbps NICs.

An interesting implementation challenge was efficiently copying gradients from intermediate network buffers to PyTorch, particularly for bucket sizes ranging from 25 MB to 500 MB in LLMs. To overcome this, we develop a custom memory copying mechanism that leverages AVX-512 streaming instructions and parallelizes the operation across multiple idle CPU cores, achieving an 8 × speedup over the default Rust `memcpy`. This optimization significantly improves performance until bandwidth becomes the limiting factor.

Model	Parameters	Parallelism	Min. CPU-nodes
ResNet50 [23]	25.6M	12 DP	1
ResNet152 [23]	60.2M	12 DP	1
ViT-H-14 [10]	633.5M	12 DP	1
GPT2-1.5B [43]	1.5B	12 DP	1
GPT3-XL [6]	1.3B	12 DP	1
GPT3-6.7B [6]	6.7B	12 DP	1
LLaMA2-7B [52]	7B	2 PP × 6 DP	2
LLaMA2-13B [52]	13B	2 PP × 6 DP	2
LLaMA3-8B [12]	8B	2 PP × 6 DP	2

Table 1: Models and configurations used in the evaluation.

We implement these functionalities using ≈ 2000 lines of Rust code to leverage Rust's safety and performance features while avoiding Python's Global Interpreter Lock (GIL) issues. We wrap the Rust code with a Python interface using PyO3 [25] to interact with PyTorch. We implement the partitioned optimizer with ≈ 1000 lines of Python code to scale out the optimizer on the Shadow nodes.

Switch: The switch is responsible for routing packets between the Shadow and training nodes. We implement the data plane functionality with ≈ 800 lines of P4 [5] and run it on a 32-port Tofino-1 Switch [2]. In addition, we implement the control plane using 1000 lines of Python code to register the replication stream, create multicast groups, and setup the TCP server. The control plane configures the switch using P4Runtime's gRPC API [20].

6 Evaluation

This section evaluates the performance, correctness, and scalability of Checkmate across various models and system scales. We begin by describing our experimental setup (§6.1), followed by a comparison against state-of-the-art checkpointing systems in terms of throughput and checkpoint frequency (§6.2). We then assess the shadow cluster's scalability, including optimizer step timing and the scaling behavior of CPU-based checkpoint nodes (§6.3, §6.4). Next, we validate the correctness of checkpointing (§6.5) and show that gradient multicasting introduces no measurable network overhead (§6.6). Finally, we quantify Checkmate's GPU-hour savings at scale under varying cluster sizes and failure rates (§6.7).

6.1 Experimental Setup

Our evaluation uses a total of 16 machines, including 12 training nodes equipped with Nvidia A100 80 GB GPUs, and up to 4 Intel Xeon Gold 5420+ shadow nodes with 28 cores each. All of these machines are connected through a 32-port Tofino-1 programmable switch operating at 100 Gbps. The training machines are set up with CUDA 12.6, PyTorch 2.6.0, and modified NCCL v2.22.3 with tagging functionality, and the shadow nodes run the same PyTorch version.

We evaluate Checkmate using various models, including ResNet and ViT for vision tasks, and GPT and LLaMA for language tasks. Smaller vision and GPT models are trained using pure DP, while each LLaMA DP replica is split 2-way

via pipeline parallelism. Table 1 shows the details for each model. We train vision models on the ImageNet dataset [9] with FP32 precision, while language models are trained on the C4 dataset [44] with BF16 precision. All the models are trained using the AdamW optimizer with a cosine annealing learning rate schedule. The vision models use an unmodified `torchvision` library for the training. For language models, we extend TorchTitan [34] to support hybrid data and pipeline parallelism, allowing evaluation across a wider set of models.

6.2 Comparison with Other Systems

We evaluate Checkmate against state-of-the-art production systems and research prototypes for models listed in Table 1. To isolate checkpointing effects, Checkmate and the No Checkpoint baseline utilize our DPDK-based network stack, whereas other systems use NCCL's default RoCE-v2 stack. We set up in-memory `nullfs` [1] across all systems to prevent persistent storage writing from becoming a bottleneck. We run 1000 training iterations for each configuration.

Figure 6 shows the result. The x-axis in each sub-figure represents the number of checkpoints over 1000 iterations, and the y-axis indicates throughput. The title on each sub-figure displays the model, local batch size, and sequence length, if applicable.

Each marker represents the throughput and checkpoint count for a given system. We also report the saved repeated work upon failure, derived from the checkpointing frequency. For example, if a given system checkpoint occurs every 6 iterations, then upon failure, it needs to repeat 3 iterations of work on average. Since Checkmate checkpoints per iteration, it always incurs 0.5 iterations of repeated work for each failure. Therefore, Checkmate needs to do $(3-0.5)/3 = 83\%$ less repeated work compared to this base system.

No Checkpoint. This baseline serves as an upper bound on throughput, indicating the maximum achievable training speed when checkpointing is disabled. Checkmate matches this baseline across all workloads, achieving virtually identical throughput. While achieving this performance, Checkmate also produces checkpoints every iteration, resulting in minimal repeated work in the event of a failure. This establishes the zero performance overhead nature of Checkmate.

We also use a No Checkpoint (DPDK) baseline that uses our custom plugin for the transport. Although the DPDK implementation consumes more CPU cores than RDMA, its performance across all configurations remains comparable.

Torch Async [41, 42]. The next comparison system is the asynchronous PyTorch Distributed Checkpoint (Torch Async). Besides asynchronously copying the checkpoint state to local CPU memory and writing it to a local `nullfs`, Torch Async shards checkpoint states across training nodes, overall reducing the overhead. We vary the checkpointing frequency (f) to evaluate trade-offs between checkpoint frequency and training throughput overhead for this system.

At $f = 20$, Torch Async yields a low overhead, only 5% slower in throughput compared to Checkmate on average. However, Torch Async needs to perform 95% more repeated work for each failure due to producing checkpoints at only 1/20 frequency of Checkmate. In contrast, Checkmate outperforms Torch Async at higher checkpointing frequencies. For instance, Checkmate achieves 2.2× and 6.5× throughput on average for vision and language models, respectively, when checkpointing per iteration. Checkmate outperforms PyTorch Async on both fronts: zero checkpoint overhead and minimized repeated work.

CheckFreq [38]. On the high level, CheckFreq shares the same goal as Checkmate: maximizing checkpoint frequency while minimizing overhead. However, CheckFreq still relies on the copy-persist framework and performs such optimization by profiling the checkpoint overhead and tuning the checkpoint frequency accordingly. It performs asynchronous checkpointing and optimizes the copy step by copying intermediate states to GPU VRAM when the models are small.

Checkmate consistently surpasses CheckFreq in both throughput and checkpoint frequency, regardless of model size. For instance, for ResNet152 (Figure 6 top left), where CheckFreq copies to GPU memory before persisting, Checkmate achieves 1.05× throughput and 15.3× more frequent checkpointing, indicating 91.5% less repeated work for each failure. For LLaMA2-13B (Figure 6 bottom left), a model with high GPU memory utilization, Checkmate achieves 1.1× throughput while checkpointing 34.5× more frequently (97.1% less repeated work per failure).

This result also illustrates the difficulty in choosing the right checkpoint frequency with profiling in practice: for GPT3-6.7B, LLaMA2-7B, and ViT-H-14, CheckFreq's initial profiling underestimated the overhead of checkpointing, resulting in more checkpoints but more overhead compared to the other models. Checkmate avoids this problem by offloading checkpointing to the network and the shadow cluster.

Gemini [56]. Like Checkmate, Gemini seeks to achieve per-iteration checkpointing by checkpointing into the remote CPU memory of the training cluster using GPUDirect-RDMA, avoiding storage stalls. It shards checkpoint states across training nodes, replicates these shards for reliability, and schedules checkpoint traffic to interleave with training without interfering with computation. In this evaluation, Gemini used the default replication factor of one. Note that Gemini employs DeepSpeed ZeRO-3, which adds an extra AllGather per iteration and increases the base iteration time compared to the other systems in our evaluation. Specifically, we exclude Gemini's results for LLaMA models, as the difference in parallelization (DP+PP hybrid vs. ZeRO-3) makes it difficult to draw fair comparisons.

Checkmate achieves higher throughput than Gemini across all models. On average, Checkmate delivers 2.2× throughput while maintaining the same checkpoint frequency for the

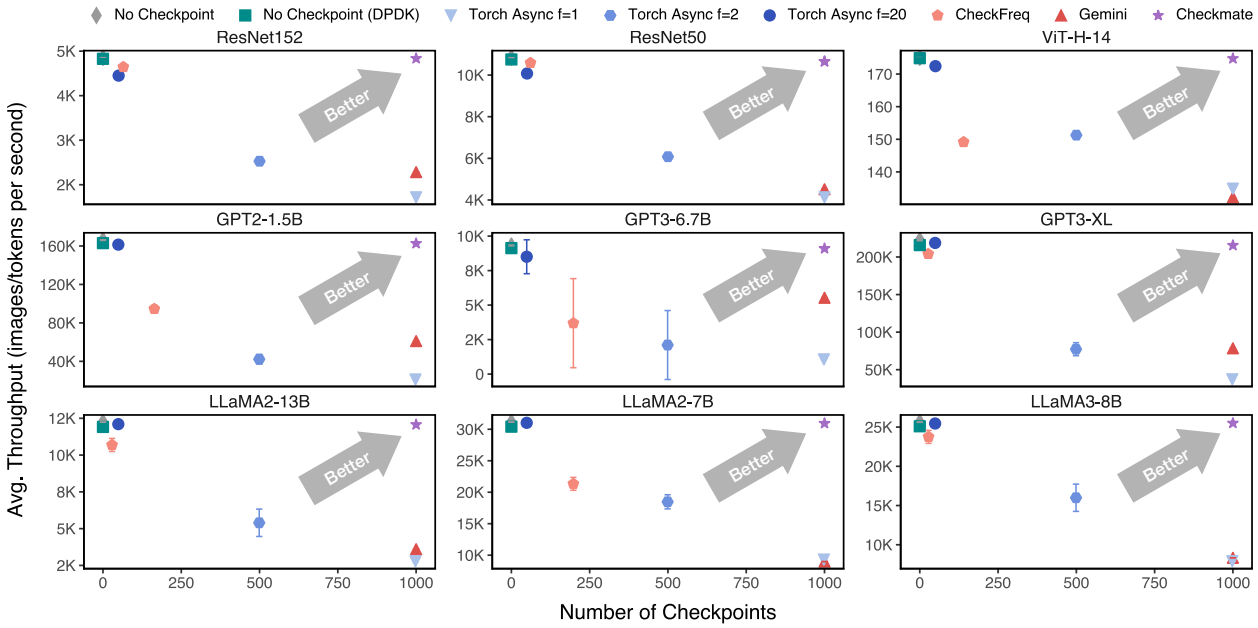


Figure 6: Training throughput and number of checkpoints for different systems. Checkmate checkpoints 5 to 34.5× more compared to CheckFreq and achieves 1.32× to 2.7× throughput than per-iteration checkpointing in Gemini on average.

vision and GPT models. Since Gemini relies on the training network to transmit data, models like ResNet with fast iteration provide less opportunity to overlap checkpointing traffic and gradient transmission, hindering its performance, especially for small batch sizes. Its overhead will also increase with the replication factor, presenting a tradeoff in performance and reliability. Checkmate avoids these problems by utilizing a dedicated network and CPU resources.

Summary. Across all models, Checkmate consistently delivers per-iteration checkpointing at no cost to training throughput. As a result, Checkmate achieves the best normal-case performance and the least repeated work upon failure.

6.3 How Fast can Checkmate Update?

The CPU-only shadow nodes must complete their optimizer step before the GPU-based training nodes begin the next one to keep pace with the GPU training nodes. We stress-test this timing constraint by sweeping the local batch size from one to the point of GPU memory saturation across both vision and GPT models. Smaller batch sizes shorten the training iteration time, pressuring the shadow nodes to finish more quickly. In contrast, larger batch sizes increase the iteration time, but a smaller number of shadow nodes must be used to take full advantage of them. To push the limits even further, this experiment uses AdamW, a widely used but computationally demanding optimizer. For every model and batch size, we present the minimum number of nodes required to complete the optimizer step and the time taken for each configuration. Additionally, we measure the average iteration time and the time spent by shadow nodes on pulling gradients and applying the optimizer step for each configuration.

Figure 7 shows the results. The solid line depicts the total iteration time, and the dashed line indicates the time spent pulling gradients. The shaded region shows the time spent applying the optimizer step. The x-axis represents the local batch size, and the y-axis shows the iteration time in milliseconds. Vertical dotted lines mark the number of shadow nodes used for each configuration. We run each experiment for 200 iterations and average the results to minimize transient effects.

As expected, the iteration time increases with batch size for all models (solid blue line). Interestingly, the time spent by shadow nodes pulling gradients (yellow dotted line) also increases with batch size, even though the total gradient size remains constant for a given model. This is because larger batch sizes lead to longer gradient computation times on the training nodes. During this time, shadow nodes remain idle, waiting for gradients to arrive, which increases the time spent pulling gradients from the network.

For the vision models (top row in Figure 7), the ratio of time spent by the shadow nodes pulling gradients to the optimizer step time is low, resulting in a minor shaded region. ResNet50 and ResNet152, being smaller models, have shorter optimizer step times. On the other hand, ViT-H-14 has a longer optimizer step time; the iteration time remains the dominant factor, which keeps the shaded region minimal.

For language models (bottom row in Figure 7), the optimizer step time is significantly longer than for vision models due to their higher compute requirements, resulting in a larger shaded region. Despite this, the shadow nodes keep up with the training nodes, often requiring only one server for larger

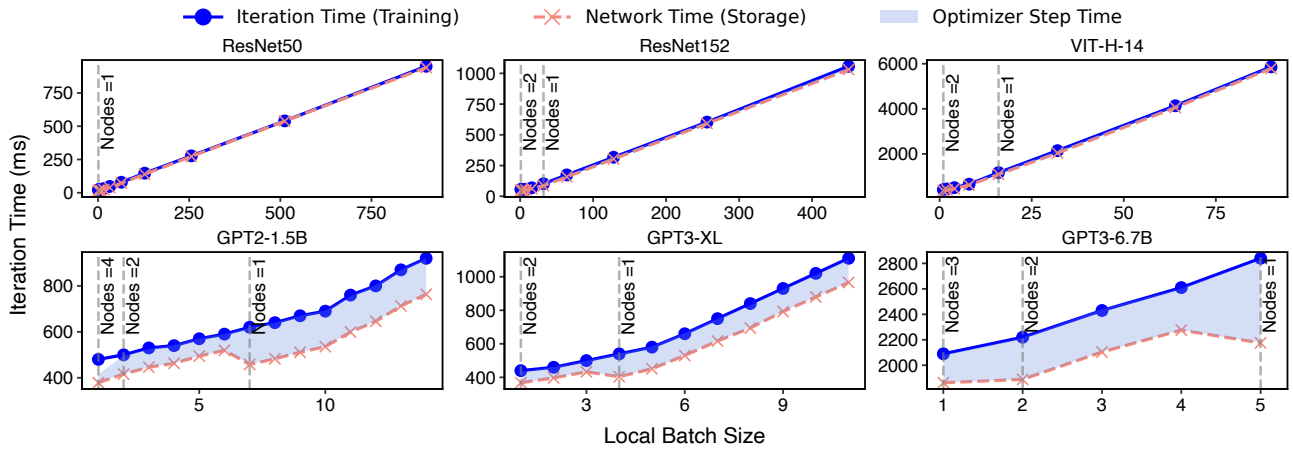


Figure 7: Impact of shadow node scaling on checkpointing time for different models. With a single shadow node, Checkmate achieves checkpointing times as low as 25 ms for smaller models (ResNet50), and completes in under 600 ms even for LLMs (GPT2-1.5B and GPT3-XL).

batch sizes. For example, Checkmate uses a single server for GPT2-1.5B, GPT3-XL, and GPT3-6.7B, with batch sizes of 7, 4, and 5, respectively. When a server is removed (e.g., at the batch size 7 for GPT2-1.5B), the shaded region expands, reflecting longer optimizer step times due to fewer available cores; however, shadow does not become a bottleneck, as the total shadow time consistently remains below the solid line.

6.4 Optimizer Scaling across Multiple Nodes

In this experiment, we analyze the impact of the number of CPU cores on the optimizer step time across various models. We vary the number of cores and measure the time it takes to run the optimizer step. Each shadow node has 28 cores, and we add an additional shadow node for every 28 cores to partition and distribute the optimizer step workload. We wrap each model with our stateless optimizer and run optimizer steps over randomized gradients.

Figure 8 demonstrates that the optimizer time decreases sharply as we add more cores, even within a single shadow node. For instance, the optimizer step time for GPT3-6.7B decreases from 6000 ms with a single core to 500 ms with 28 cores. The step time decreases almost linearly as we add more shadow nodes, with each vertical dotted line representing the addition of a new shadow node. For the same example, with 56 cores, the step time decreases to 350 ms, and with all 112 cores, it decreases to 180 ms.

For most models in our evaluation, two shadow nodes are sufficient to keep up with training. For example, with GPT3-6.7B, the iteration time for a local batch size of 4 is greater than 2000 ms, but even with half the cores on a single shadow node, the optimizer step time is less than 1000 ms. This highlights that even for the large model, Checkmate only needs a few shadow nodes, equipped with a typical datacenter server CPU, to keep up with the training nodes.

6.5 Checkpoint Correctness

We investigate whether the model and optimizer states are mathematically equivalent across the training and shadow clusters. Ensuring this is crucial to verify that the training process remains unaffected by the checkpointing mechanism. To verify this, we train a vision model (ResNet152) over the ImageNet dataset for 500 iterations over four training nodes and one shadow node. We repeat the training process twice using the same random seed and identical hyperparameters. In the first run, training proceeds uninterrupted, and we record the training loss after each iteration. In the second run, we intentionally halt training during every second iteration, restoring the model weights and optimizer states from the shadow nodes before resuming the next iteration.

Figure 9 shows the loss for ResNet152. The solid line represents the uninterrupted run, and the dashed line represents the interrupted run. The lines overlap completely, indicating that the training loss over time is identical. This observation confirms that Checkmate's checkpointing mechanism preserves the mathematical equivalence of the training process.

To further validate these findings, we perform a third experiment. In this case, we compare the model weights, biases, and optimizer states, such as momentum and learning rate, up to the 8th decimal place between the training and shadow clusters after each iteration. The comparison shows no discrepancies, reinforcing the conclusion that the Checkmate maintains checkpointing correctness.

6.6 Network Overheads of Multicasting

Checkmate uses the network switch to efficiently multicast tagged gradients without slowing training. Tofino-1 has a dedicated Packet Replication Engine (PRE) capable of multicasting at full line rate. We want to ensure that the switch can operate at line rate, even with a higher replication factor.

To verify this, we run the NCCL AllReduce test on 1 GB of

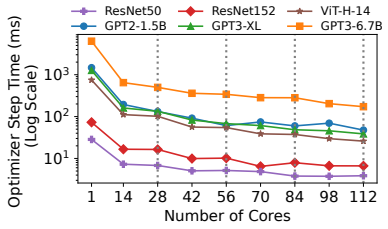


Figure 8: Checkmate's optimizer step time scales linearly with the number of nodes.

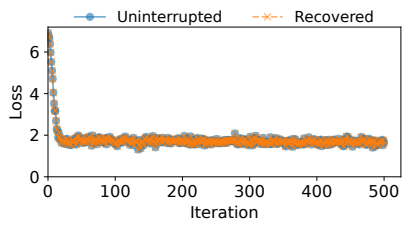


Figure 9: The uninterrupted and recovered models converge identically (ResNet152).

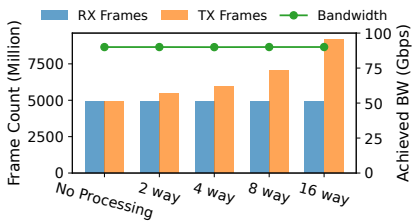


Figure 10: Achieved bandwidth and frame count for different replication factors.

data for 200 iterations across four training nodes and replicate the tagged gradients to 2 to 16 additional switch ports. We measure the AllReduce bus bandwidth and the total transmitted frames, as counted by the switch. For comparison, we evaluate the baseline throughput without packet replication.

Figure 10 shows the bus bandwidth and transmitted frames. The RX frame count and the total data transmitted from the GPUs remain the same across all replication factors. Given that the switch only replicates tagged packets, an increase in TX does not proportionally increase the network load on the switch. Even at 16-way replication, the switch only transmits 1.9× the number of frames it receives. Note that the switch does not replicate all the packets, but only the tagged ones. In all cases, the AllReduce bus bandwidth remains constant.

6.7 GPU Savings at Scale

The prior section evaluated Checkmate's feasibility and performance on a local testbed. This section extends the analysis to larger scales and varying failure rates with simulation.

Figure 11 shows the GPU hours saved by Checkmate compared to traditional checkpointing systems. We assume a training workload similar to LLaMA3-405B, with an iteration time of 4.58 seconds (Appendix A). Based on the checkpoint overhead, we apply the optimal checkpoint frequency that minimizes the expected waste of GPU hours (Appendix B).

For each subplot, the x-axis sweeps checkpoint overhead values to reflect a range of implementations, from synchronous to various asynchronous checkpointing, while the y-axis illustrates the expected GPU hours saved per day by Checkmate. Each line corresponds to a different cluster size, and each subplot represents different GPU failure rates. The right-hand side subplot uses Meta's reported failure rate from LLaMA3 training at 2.0×10^{-5} failures per GPU-hour.

Three key observations emerge from this analysis. First, Checkmate yields greater savings as the cluster size increases. Checkmate saves 16 times more GPU hours at 16,384 GPUs compared to 4,096 GPUs for both subfigures. This stems from the quadratic increase in wasted work with system scale, as derived in Appendix B. Second, Checkmate provides meaningful benefits even under low checkpoint overhead. The right-hand side figure shows that assuming just 10 ms overhead per checkpoint (lowest point on x-axis), Checkmate still saves 448 GPU-hours per day at the 16,384-GPU scale (blue line).

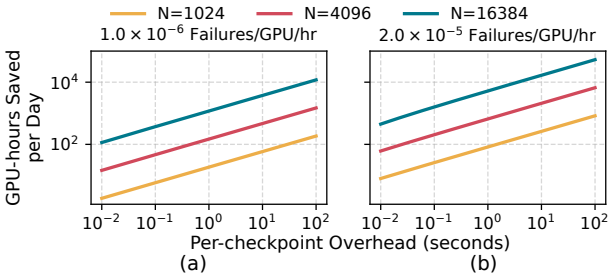


Figure 11: GPU hours saved per day by Checkmate compared to traditional checkpointing systems across varying cluster scales, checkpoint overheads, and GPU failure rates. Checkmate yields increasing savings at larger scales, remains effective under low checkpoint overhead, and remains beneficial at low failure rates.

Over a 54-day training period, this results in more than 24,000 GPU-hours saved. Third, Checkmate remains effective even if failure rates improve significantly. The left-hand side figure shows that at a failure rate of 10^{-6} failures per GPU-hour (0.5% of Meta's observed rate), Checkmate still saves nearly 70,000 GPU-hours across a 54-day training run.

7 Related Work

Checkmate builds on a rich body of work in distributed training, model checkpointing, and in-network computation. Throughout the paper, we have discussed prior efforts on improving checkpointing mechanism; in this section, we discuss the most relevant work in the rest of each area.

Alternative to checkpointing systems. Most existing systems resume training by restoring from explicitly stored checkpoints. In contrast, recent work explores mechanisms that avoid conventional checkpointing and instead introduce alternative ways to resume training after failures. JIT Checkpointing [22] eliminates periodic checkpointing by reconstructing the minimal state needed to resume training only upon failure, thereby removing checkpointing overheads at the cost of additional computation during recovery. ByteCheckpoint [53] and Universal Checkpointing [33] are orthogonal efforts that redesign checkpointing around a uniform, parallelism-agnostic representation, enabling efficient resizing across heterogeneous training layouts. However, they still rely on periodic checkpointing, which stall training for large-scale models. Unlike Checkmate, neither system can fully remove training stalls.

Fault-tolerance using redundant computation. Redundant computation represents another class of fault-tolerance techniques in distributed training. Systems like Bamboo [51], Oobleck [26], and ReCycle [14] use redundant computation to handle stragglers and failures. While effective, these approaches incur high computational and communication overhead due to duplicate computations. In contrast, Checkmate minimizes overhead by using minimal CPU resources to maintain shadow models and optimizer states, which are only accessed during recovery, offering a lightweight and efficient alternative for fault tolerance.

Optimizer offloading and sharding. Systems like DeepSpeed ZeRO-Offload [45, 46], leverage CPU offloading to reduce GPU memory usage and utilize CPU computing, enabling the training of larger models. In addition to offloading computations, these systems commonly use CPU memory to store optimizer states [38,56], activations [30], and weights. Systems like DeepSpeed [46] and NVIDIA NeMo [30] effectively use CPU memory for these purposes to optimize resource usage. Techniques such as ZeRO, FSDP, and OSDP [28,45,58] shard the optimizer state across multiple devices mainly to reduce memory usage. Checkmate builds on optimizer offloading and sharding techniques to enable efficient checkpointing.

Optimizing DNN training with in-network computation. In-network computing optimizes distributed DNN training by offloading computations to network infrastructure. Systems like SwitchML [48], ATP [31], SHARP [19], and PANAMA [16] use programmable switches to perform gradient aggregation, reducing communication overhead. While these systems focus on optimizing gradient communication, Checkmate complements them by capturing gradients during training and transmitting them to a shadow cluster for checkpointing. Unlike aggregation systems, Checkmate does not require specialized switch hardware, such as floating point units, making it more widely applicable.

Lossless delivery for datagrams. Checkmate relies on lossless packet delivery for correct gradient replication. While this requirement appears stringent, large-scale production systems have demonstrated the feasibility of deploying lossless networks in real-world settings. For instance, Meta's DNN training clusters [15] and Microsoft's Azure Storage [4] achieve reliable lossless delivery at scale. In Meta's LLaMA3 training infrastructure, mechanisms like DCQCN are disabled, and PFC serves as the sole method for flow and congestion control, even at scales involving 16K GPUs [12]. Although we do not evaluate Checkmate at such a scale, it leverages the same PFC mechanism, indicating its potential for scalability. Additionally, recent research has shown that lossless delivery is feasible even in high-demand scenarios. For example, BFC [18] demonstrates reliable lossless transmission in data center networks with high-degree incast scenarios (e.g., 2000-to-1), further supporting the deployability of lossless delivery mechanisms for large-scale systems.

8 Discussion

Applicability to other AllReduce algorithms. While Checkmate's gradient replication uses Ring-AllReduce (§ 4.1), it is easy to extend to other algorithms. AllReduce operations generally follow one of two patterns: a ReduceScatter followed by an AllGather (e.g., Ring-AllReduce, as used in our implementation) or a Reduce followed by a Broadcast (e.g., the double binary tree algorithm [27]). In both cases, the network transmits reduced gradients, enabling Checkmate to identify and replicate gradients to the shadow cluster.

Checkmate and FSDP training. Techniques like FSDP [58] and ZeRO [45] replace the AllReduce with a ReduceScatter for gradients and an AllGather for model parameters, which appears to be incompatible with Checkmate. However, Checkmate can extend its multicasting abstraction to support these strategies. Specifically, Checkmate multicasts model parameters in the AllGather step in FSDP to shadow servers for checkpointing, ensuring parameters are recoverable even if optimizer states are not directly captured.

For optimizer states, if the update step is linear to the gradient, we invert the update equation and solve for the gradient with new parameters, subsequently updating the optimizer state. For optimizers that track the second moment, such inversion yields a quadratic equation and, thus, two possible solutions for the gradient. In these cases, Checkmate can still function with minor modifications, such as transmitting additional metadata during the AllGather step to resolve ambiguities. We recognize that fully adapting Checkmate to support FSDP and ZeRO requires a comprehensive evaluation of the trade-offs, which we leave as future work.

Applicability in public clouds. Checkmate's design relies on programmable switches to multicast gradient traffic. However, in a cloud environment, tenants have no access to switch-level programmability, and advanced NIC capabilities are limited to certain instance types. Consequently, using Checkmate on today's public clouds is restricted by these constraints. Broader adoption will require either provider-supported tiers optimized for training with the necessary network primitives or shifting the burden to compute nodes to handle all processing at the cost of reduced network efficiency.

9 Conclusion

Checkmate introduces a novel approach to checkpointing that leverages a multicast operation to efficiently capture gradients from the network and offload checkpointing to a scalable shadow cluster. This design eliminates training pauses or slowdowns. Checkmate achieves 5 to 34.5× more frequent checkpointing compared to state-of-the-art checkpointing frameworks, resulting in 80% to 97.1% reduction in repeated work per failure. At the same checkpointing frequency, Checkmate delivers 1.3× to 6.5× throughput compared to other systems.

Acknowledgments

We thank our shepherd, Dan Li, and anonymous reviewers for their insightful suggestions and feedback. We also thank Om Chabra, Gohar Irfan Chaudhry, Josh Fried, Pouya Hamadanian, Ben Holmes, Zain Zhenyuan Ruan, and Anton A. Zabreyko, as well as other members of the Network and Mobile Systems (NMS) and Parallel and Distributed Systems (PDOS) research groups, for their helpful feedback. This research was supported by NSF SHF-2107244, NSF CAREER-2144766, NSF PPoSS-2217099, NSF CNS-2211382, NSF FuSe-TG-2235466, Sloan fellowship FG-2022-18504; and by ACE, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Michael Ablassmeier. nullfs: A Virtual File System That Behaves Like /dev/null, 2024. https://github.com/abbbi/nullfsvfs.

[2] Anurag Agrawal and Changhoon Kim. Intel Tofino2 - A 12.9Tbps P4-Programmable Ethernet Switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–32, 2020.

[3] Dario Amodei, Danny Hernandez, Girish Sastry, Jack Clark, Greg Brockman, and Ilya Sutskever. AI and compute. https://openai.com/index/ai-and-compute/.

[4] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *CoRR*, abs/2005.14165, 2020.

[7] Checkmate github repository. https://github.com/hipersys-team/checkmate, 2025.

[8] NVIDIA Corporation. NCCL Net Plugin Documentation. https://github.com/NVIDIA/nccl/blob/master/ext-net/README.md, 2025.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale. *CoRR*, abs/2010.11929, 2020.

[11] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: A Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, Renton, WA, April 2022. USENIX Association.

[12] Aaron Grattafiori et al. at Meta. The Llama 3 Herd of Models, 2024.

[13] FS.com. NVIDIA Mellanox MCX75510AAS-NEAT ConnectX-7 InfiniBand Adapter Card 400GbE/NDR. https://www.fs.com/products/177272.html?attribute=104280&id=4656075, 2025.

[14] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. ReCycle: Resilient Training of Large DNNs using Pipeline Adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating*

Systems Principles, SOSP '24, pages 211–228. ACM, November 2024.

[15] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, pages 57–70, New York, NY, USA, 2024. Association for Computing Machinery.

[16] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-Network Aggregation for Shared Machine Learning Clusters. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 829–844, 2021.

[17] Google. Google Cloud's Pricing Calculator. https://cloud.google.com/products/calculator, 2025.

[18] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure Flow Control. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 779–805, Renton, WA, April 2022. USENIX Association.

[19] Richard L. Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, Ami Marelli, Valentin Petrov, Evyatar Romlet, Yong Qin, and Ido Zemah. Scalable Hierarchical Aggregation and Reduction Protocol (SHARP)™ Streaming-Aggregation Hardware Design and Evaluation. In *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22-25, 2020, Proceedings*, pages 41–59, Berlin, Heidelberg, 2020. Springer-Verlag.

[20] gRPC Authors. A High Performance, Open Source Universal RPC Framework. https://grpc.io/, 2024.

[21] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[22] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. Just-In-Time Checkpointing: Low Cost Error Recovery from Deep Learning Training Failures. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*, pages 1110–1125, Athens, Greece, April 2024. Association for Computing Machinery.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *CoRR*, abs/1512.03385, 2015.

[24] Tao He, Xue Li, Zhibin Wang, Kun Qian, Jingbo Xu, Wenyuan Yu, and Jingren Zhou. Unicron: Economizing Self-Healing LLM Training at Scale. In *arXiv*, 2023.

[25] David Hewitt and PyO3 contributors. The PyO3 User Guide. https://pyo3.rs/v0.23.3/, 2024.

[26] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient Distributed Training of Large Models Using Pipeline Templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 382–395, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Sylvain Jeaugey. Massively Scale Your Deep Learning Training with NCCL 2.4, Feb. 2019. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/.

[28] Youhe Jiang, Fangcheng Fu, Xupeng Miao, Xiaonan Nie, and Bin Cui. OSDP: Optimal Sharded Data Parallel for Distributed Deep Learning, 2023.

[29] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[30] Oleksii Kuchaiev, Jason Li, Huyen Nguyen, Oleksii Hrinchuk, Ryan Leary, Boris Ginsburg, Samuel Kriman, Stanislav Beliaev, Vitaly Lavrukhin, Jack Cook, Patrice Castonguay, Mariya Popova, Jocelyn Huang, and Jonathan M. Cohen. NeMo: A Toolkit for Building AI Applications Using Neural Modules. *CoRR*, abs/1909.09577, 2019.

[31] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, April 2021.

[32] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *Proc. VLDB Endow.*, 13(12):3005–3018, August 2020.

[33] Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, Masahiro Tanaka, Stas Bekman, Olatunji Ruwase, and Minjia Zhang. Universal Checkpointing: A Flexible and Efficient Distributed Checkpointing System for Large-Scale DNN Training with Reconfigurable Parallelism. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, USA, 2025. USENIX Association.

[34] Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. TorchTitan: One-Stop PyTorch Native Solution for Production Ready LLM Pre-Training, 2024.

[35] Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization. *CoRR*, abs/1711.05101, 2017.

[36] ByteDance Ltd. Transport Protocol Acceleration Library. https://bytedance.github.io/libtpa/, 2023.

[37] Avinash Maurya, Robert Underwood, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. DataStates-LLM: Lazy Asynchronous Checkpointing for Large Language Models. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '24)*, pages 227–239, Pisa, Italy, June 2024. Association for Computing Machinery.

[38] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 203–216. USENIX Association, February 2021.

[39] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 172–181, 2020.

[41] pytorch.org. Distributed Checkpoint - torch.distributed.checkpoint, 2024. https://pytorch.org/docs/stable/distributed.checkpoint.html.

[42] pytorch.org. TorchSnapshot - Getting Started, 2024. https://pytorch.org/torchsnapshot/stable/getting_started.html.

[43] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language Models Are Unsupervised Multitask Learners. *OpenAI blog*, 1(8):9, 2019.

[44] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[45] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2020.

[46] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pages 3505–3506, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Sebastian Ruder. An Overview of Gradient Descent Optimization Algorithms. *CoRR*, abs/1609.04747, 2016.

[48] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling Distributed Machine Learning with In-Network Aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[49] shopblt.com. H100 P4387 SYSTEM 640GB. https://www.shopblt.com/item/nvidia-dgx-dgxh-g640fp2edi60-h100-p4387-system/dvd6_dgxhg640fp2edi60.html, 2025.

[50] thinkmate.com. SuperMicro SuperServer 511E-WR. https://www.thinkmate.com/system/superserver-511e-wr, 2025.

[51] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, Boston, MA, April 2023. USENIX Association.

[52] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models, 2023.

[53] Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '25)*, pages 559–578, Philadelphia, PA, USA, April 2025. USENIX Association.

[54] Guanhua Wang, Olatunji Ruwase, Bing Xie, and Yuxiong He. FastPersist: Accelerating Model Checkpointing in Deep Learning, 2024.

[55] Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani. Rail-only: A Low-Cost High-Performance Network for Training LLMs with Trillion Parameters. In *2024 IEEE Symposium on High-Performance Interconnects (HOTI)*, pages 1–10, Los Alamitos, CA, USA, August 2024. IEEE Computer Society.

[56] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 364–381, New York, NY, USA, 2023. Association for Computing Machinery.

[57] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models, 2022.

[58] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel, 2023.

A Iteration and Checkpoint Time of LLaMA3

In the LLaMA3 technical report [12], Meta did not specify the training iteration time for each phase, rendering it hard to estimate the repeated work. However, with other published data, it is possible to calculate their training iteration time, which we report below.

We estimate the training iteration time by calculating the FLOPs (floating-point operations) for a single LLaMA training iteration. We start by estimating the FLOPs required for a single LLaMA-style Transformer model forward pass employing Grouped Query Attention (GQA). The key operations considered include attention projections, attention computations, feed-forward networks (FFN), rotary positional embeddings (RoPE), and the final language modeling head projection. We provide a succinct overview of each component and the formula for estimating its computational cost. Table 2 summarizes the notation we use in our derivation.

Symbol	Meaning
b	Batch size
s	Sequence length
L	Number of Transformer layers
h	Hidden dimension
f	Feed-forward (FFN) dimension
v	Vocabulary size
a	Total number of attention heads for queries (Q)
g	Number of groups for keys (K) and values (V) in GQA

Table 2: Notation used in FLOPs calculation

Attention Projections (QKV). Each Transformer layer computes linear projections to obtain queries (Q), keys (K), and values (V). We calculate the cost as:

$$\text{FLOPs}_{\text{QKV}} = 2\left(b \cdot s \cdot h^2 + 2 \cdot b \cdot s \cdot h \cdot (g \cdot a)\right)$$

Attention Computation. The dot-product attention and the subsequent multiplication with V incur:

$$\text{FLOPs}_{\text{Attn}} = 4 \cdot b \cdot s^2 \cdot h$$

Attention Output Projection. Projecting attention outputs back to the hidden dimension costs:

$$\text{FLOPs}_{\text{AttnOut}} = 2 \cdot b \cdot s \cdot h \cdot (g \cdot a)$$

Feed-Forward Network (FFN). Each FFN layer includes two linear transformations:

$$\text{FLOPs}_{\text{FFN}} = 4 \cdot b \cdot s \cdot h \cdot f$$

Rotary Positional Embedding (RoPE). Positional embeddings add a computational overhead, approximated by:

$$\text{FLOPs}_{\text{RoPE}} = 2 \cdot b \cdot s \cdot h$$

The total FLOPs per Transformer layer combine all these components:

$$\begin{aligned}\text{FLOPs}_{\text{Layer}} = {} & \text{FLOPs}_{\text{QKV}} + \text{FLOPs}_{\text{Attn}} \\ & + \text{FLOPs}_{\text{AttnOut}} + \text{FLOPs}_{\text{FFN}} + \text{FLOPs}_{\text{RoPE}}\end{aligned}$$

Accumulation over Layers. Summing across L layers:

$$\text{FLOPs}_{\text{TotalLayers}} = \text{FLOPs}_{\text{Layer}} \cdot L$$

Final LM Head Projection and word embedding. Projecting hidden states to the vocabulary dimension, and projecting the vocabulary into the hidden dimension costs:

$$\text{FLOPs}_{\text{Vocab}} = 4 \cdot b \cdot s \cdot h \cdot v$$

Total FLOPs. The final estimated FLOPs for a single forward pass are:

$$\text{FLOPs}_{\text{Total}} = \text{FLOPs}_{\text{TotalLayers}} + \text{FLOPs}_{\text{Vocab}}$$

The total FLOPs of an iteration is roughly three times the FLOPs of one forward pass, since the backward pass performs the same amount of operations for both the activation gradients and the weight gradients. Note that the LLaMA3 technical report specified that activation checkpointing is disabled. Therefore, the backward pass does not enclose another forward pass of computation.

We validate our formulation by computing the total FLOP for LLaMA training with Meta's reported numbers and comparing it to the 3.5×10^{25} FLOPs training budget specified by LLaMA. Accounting for all three phases in pretraining, our formula yields a total training FLOP of 3.49×10^{25}, which closely matches the published number.

With this estimation, the training iteration time is

$$\text{Iteration time} = \frac{\text{FLOPs}_{\text{Total}}}{\text{Achieved FLOP per GPU} \times \text{Total GPUs}} \tag{1}$$

For the major phase of LLaMA3-405B training with a batch size of 16M, sequence length of 8192, and total GPU count of 16384, Meta achieved 400 TFLOPs per-GPU [52]. Using our formulas above, the estimated iteration time is 4.58s.

To estimate the checkpointing time, we use numbers reported in the LLaMA3 technical report, where Meta mentions that their checkpoint storage cluster has a sustainable throughput of 2 TB/s. For the 405B parameter model, the total checkpoint size is

$$\begin{aligned}&\text{405B parameters} \times \\ &(\text{2 bytes per parameter} + \text{4 bytes per optimizer state}) \\ &= \text{2.43 TB}\end{aligned}$$

Hence, the checkpoint process takes 1.2s.

B Cost Analysis

In this section, we analyze the operational cost of Checkmate. We seek to understand the operational scheme of Checkmate: given a model and a checkpoint overhead, how does scale and failure rate impact Checkmate's effectiveness in saving?

B.1 Cost Modeling

We utilize cloud pricing for GPUs and CPUs and model the associated costs. Table 3 summarizes the variables we use in this derivation.

Variable	Meaning
λ	Failure rate of GPUs (per hour)
N	Number of GPUs
D	Total training duration without failure
f	Checkpoint frequency
t	Iteration time
ω	Checkpoint stall time
C	Number of CPU nodes required for Checkmate
P_G	GPU price ($/GPU/hour)
P_C	CPU node price ($/CPU-Node/hour)

Table 3: Notation used in cost calculation.

In existing checkpointing systems, the total wasted GPU hours consist of the repeated work from GPU failure and the checkpointing overhead. Checkpointing happens every ft time. Assuming a constant failure rate, a failure occurs uniformly between checkpoints. Therefore, the expected repeated work is half of the checkpoint interval for each failure $\frac{ft}{2}$. For an N-GPU system, assuming GPU failures are IID, then the failure follows a binomial distribution. Therefore, the expected number of failures is λND. Thus, the expected wasted total GPU hour during the entire training duration is $\frac{1}{2}\lambda N^2 Dft$.

The checkpoint overhead depends on the checkpointing frequency f. Assume each checkpoint stalls the training with time ω. For simplicity, we assume the per-checkpoint overhead is a constant in this derivation, though we note that as checkpoint frequency goes up, this overhead increases due to the copy and persist step contending each other more. Then, the total checkpoint overhead throughout the entire training is $\frac{ND}{ft}\omega$. The sum of both terms provides the total wasted hour:

$$\text{Wasted}_{SOTA}(f) = ND\left(\frac{1}{2}\lambda Nft + \frac{\omega}{ft}\right) \tag{2}$$

This is the equation for the GPU-hour curve in Figure 1. For a per-hour GPU price P_G, the total cost of wasted GPU hours, as a function of checkpoint frequency, is

$$\text{Cost}_{SOTA}(f) = P_G ND\left(\frac{1}{2}\lambda Nft + \frac{\omega}{ft}\right) \tag{3}$$

For Checkmate, we require C CPU-nodes to update the checkpoint together with training consecutively. Therefore, Checkmate will spend DC total CPU-node hours and achieve a per-iteration checkpoint. Checkmate still faces, on average, half an iteration of repeated work, which amounts to $\frac{1}{2}N^2 D\lambda t$ GPU hours. Hence, Checkmate's total cost of keeping checkpointing plus the cost of wasted GPU hours is

$$\text{Cost}_{\text{Checkmate}} = \frac{1}{2}P_G\lambda N^2 D + P_C DC \tag{4}$$

Checkmate becomes the optimal solution for checkpointing when

$$\text{Cost}_{\text{Checkmate}} < \text{Cost}^*_{SOTA} \tag{5}$$

Where Cost^*_{SOTA} is the minimum cost of a conventional system for *any* checkpoint frequency. Taking the derivative of $\text{Cost}_{SOTA}(f)$, we find the optimal checkpoint frequency that minimizes cost. The frequency is also at least one, since it is infeasible to checkpoint fractional iterations. Therefore, the optimal checkpoint frequency is

$$f^* = \begin{cases} \sqrt{\frac{2\omega}{\lambda N t^2}}, & \sqrt{\frac{2\omega}{\lambda N t^2}} \geq 1 \\ 1, & \text{otherwise.} \end{cases}$$

This gives the minimum cost

$$\text{Cost}^*_{SOTA} = \begin{cases} P_G ND\sqrt{2\omega\lambda N}, & \sqrt{\frac{2\omega}{\lambda N t^2}} \geq 1 \\ P_G ND\left(\frac{1}{2}\lambda Nt + \frac{\omega}{t}\right), & \text{otherwise.} \end{cases}$$

To estimate the total cost of repeated work, we utilize the public cloud pricing of H100 SXM5 GPUs, the same GPU used in Meta's LLaMA3 training cluster, on Google Cloud, leveraging Google's cloud platform cost estimator [17]. At the time of writing, H100 GPU costs $11.06 per hour, while a CPU node with 32 cores and 128 GB of DRAM costs $1.28 per hour.

We calculate the CPU time for Checkmate with LLaMA3 assuming 128 CPU servers, each handling two multicasting streams. Since LLaMA3 training run for 54 days, the total CPU-node hours are 54 days $\times$ 128 $\times$ days $\times$ 24 hours/day = 166K CPU-node hours.

B.2 Case Study

We analyze the cost-effectiveness of Checkmate compared to traditional checkpointing as GPU failure rates and checkpoint overhead vary. Our analysis utilizes infrastructure and training parameters published by Meta for the LLaMA3-405B model, evaluating the training of a 405 billion-parameter model on a cluster of 16,384 Nvidia H100 SXM5 GPUs. The model assumes an iteration time of 4.58 seconds (Appendix A) with 128 data-parallel shards.

Conventional checkpointing systems incur costs from two primary factors: repeated computations resulting from GPU failures and periodic checkpoint stalls. Checkmate, however, performs checkpointing every iteration without stalls by using additional CPU nodes concurrently with GPU training, thus trading GPU recomputation hours for CPU hours. We convert the wasted GPU hours and CPU hours spent on checkpointing into monetary terms using current cloud pricing as estimates.

Figure 12 illustrates the cost comparison between Checkmate and traditional checkpointing across different GPU failure rates and checkpoint overhead scenarios. We consider checkpoint overheads ranging from 10 ms to 2 seconds per checkpoint. Our results indicate scenarios where Checkmate

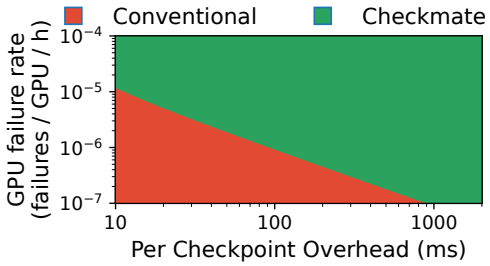


Figure 12: Solution space for Checkmate and conventional checkpointing systems for minimizing cost due to failure, at 16K GPU scale for the LLaMA3-405B model. At current reported failure rate of $\sim 10^{-5}$ per GPU-hour, Checkmate outperforms traditional systems even under optimal assumptions of 10 ms per checkpoint overhead.

achieves better cost efficiency in the green region of the plot. At the 16K GPU scale, Checkmate becomes cost-effective when the GPU failure rate exceeds approximately 1.1×10^{-7} failures per GPU-hour, assuming Meta's reported checkpoint overhead of 1.2 seconds. This threshold is substantially lower than Meta's observed GPU failure rate of 2.0×10^{-5} per GPU-hour in LLaMA3 training or 4.9×10^{-5} per GPU-hour in OPT training [12, 57], demonstrating Checkmate's immediate practical advantage under current conditions. Even if we consider a highly optimistic checkpoint overhead of just 50 ms, Checkmate remains cost-effective at GPU failure rates exceeding 2.7×10^{-6} per GPU-hour.

If future advancements significantly reduce GPU failure rates (from 10^{-5} to 10^{-7}), Checkmate still maintains a performance advantage, but its cost advantage diminishes. In such scenarios, further optimization of Checkmate, such as reducing the CPU-node overhead, becomes crucial to sustain its economic benefits.