CheckWork: Enabling Trace-Driven Analysis of Checkpointing Overhead in Distributed ML Training

Eldar Hasanov
Imperial College London
London, United Kingdom
eldar.hasanov@imperial.ac.uk

Adel Sefiane
Imperial College London
London, United Kingdom
NVIDIA
Roskilde, Denmark
asefiane@nvidia.com

Alireza Farshin
NVIDIA
Stockholm, Sweden
afarshin@nvidia.com

Marios Kogias
Imperial College London
London, United Kingdom
m.kogias@imperial.ac.uk

Abstract

Checkpointing is a fundamental mechanism for fault tolerance in large-scale distributed machine learning (ML) training, but it can introduce significant overhead due to interactions with compute and communication phases. Evaluating checkpointing strategies on production clusters is costly, difficult to reproduce, and limits the ability to systematically explore alternative designs. We present CheckWork, a framework that generates checkpoint-aware Chakra execution traces by augmenting training DAGs with checkpoint operations. These traces enable realistic simulation of checkpointing strategies using existing system simulators. Our evaluation shows that the generated traces reproduce checkpointing behaviour previously observed on real test-beds and reported in systems such as CheckFreq and Gemini, capturing both computational overheads and network-level communication interference. Using the traces, we also analyse interference between checkpoint traffic and training communication in multi-job environments and propose a mechanism to mitigate this effect.

CCS Concepts

• **Networks** → **Network simulations**; **Data center networks**; • **Computing methodologies** → Distributed artificial intelligence; **Distributed algorithms**.

Keywords

Checkpointing, Synthetic Chakra Execution Traces, Simulation

ACM Reference Format:
Eldar Hasanov, Adel Sefiane, Alireza Farshin, and Marios Kogias. 2026. CheckWork: Enabling Trace-Driven Analysis of Checkpointing Overhead in Distributed ML Training. In *The 10th Asia-Pacific Workshop on Networking (APNet 2026), August 06–07, 2026, Singapore, Singapore.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3820441.3820476

APNet 2026, Singapore, Singapore

ACM ISBN 979-8-4007-2664-4/26/08
https://doi.org/10.1145/3820441.3820476

1 Introduction

Modern machine learning (ML) training workloads have grown dramatically in scale, with frontier models requiring thousands of GPUs operating continuously for weeks or months to complete a single training run [4, 15, 17]. As training clusters grow in size and cost, long-running jobs become increasingly vulnerable to hardware failures, software faults, and infrastructure interruptions [7, 8]. To mitigate these risks, distributed training systems periodically save the state of the training process through checkpointing, which captures model parameters along with optimiser state.

However, checkpoint operations directly interact with training's compute and communication patterns, and poorly designed checkpointing strategies can introduce significant performance overhead. Recent work reports that checkpoint saving alone accounts for up to 8.8% of training time in the evaluated workloads [11].

Prior research proposes a variety of strategies that are typically evaluated on production GPU clusters, public cloud infrastructure, or large HPC environments [13, 14, 16, 22, 24]. While such deployments provide realistic measurements, they also impose practical limitations: experiments are expensive, difficult to reproduce, and constrained to a small set of configurations. Moreover, findings obtained on a particular cluster configuration may not readily generalise to other environments, since checkpointing overheads depend strongly on the underlying infrastructure characteristics such as network capacity and storage systems. As a result, exploring the broader design space of checkpointing strategies remains challenging. This motivates the need for a framework that enables inexpensive experimentation and systematic what-if analysis.

Synthetic workload traces have recently emerged as an alternative to hardware test-beds for studying large-scale ML systems. Frameworks such as MLSynth [19], STAGE [12], and SimAI [23] generate traces from high-level model descriptions and parallelisation strategies, enabling workloads to be replayed in full-stack simulators. These traces commonly rely on the Chakra execution trace (ET) format [20], which represents ML workloads as directed acyclic graphs (DAGs).

Building on this foundation, we introduce CheckWork, an open-source framework for generating checkpoint-aware execution traces from high-level configurations. CheckWork augments the generated Chakra ETs with checkpoint operations while

preserving the original training dependencies, allowing the traces to capture a wide range of checkpointing strategies. This facilitates analysis of checkpoint-related overhead across both compute and network communication. With CheckWork, researchers can conduct large-scale design space exploration and what-if analysis of how checkpointing interacts with parameters such as parallelisation strategy, topology, scheduling, and model size. To the best of our knowledge, this is the first system that systematically produces checkpointing workloads and allows the evaluation of checkpointing approaches without requiring access to a physical test-bed. The source code of CheckWork is publicly available at https://github.com/NetMLSim/checkwork.

Contributions. In this work, we:

- Introduce CheckWork, an open-source framework that generates checkpoint-aware execution traces for distributed ML training workloads. The framework augments Chakra ETs with checkpointing operations while preserving training dependencies, enabling simulation of different checkpointing strategies.
- Validate the realism of the generated traces by reproducing results of two state-of-the-art checkpointing strategies (CheckFreq [14] & Gemini [24]), and demonstrating that trace-driven simulation can *accurately* capture checkpoint overhead trends observed in real deployments.
- Demonstrate the value of these traces by showing how they can enable *previously unexplored* analyses. In particular, we identify multi-job scenarios where checkpointing traffic can be prioritised over workload communication, contrary to prior approaches.

2 Background

2.1 Checkpointing in Distributed ML Training

Checkpointing is a widely used reliability mechanism in distributed systems, where the state of an application is periodically saved to persistent storage so that execution can resume from the last saved state after a failure [3, 22]. In the context of large-scale machine learning training, a checkpoint typically consists of the model parameters together with optimiser state and additional training metadata such as iteration counters, random number generator state, dataloader position, and scheduler configuration. These artefacts allow training to resume deterministically without restarting the entire workload. Checkpointing systems can be designed along several dimensions:

(1) **Synchronous vs. asynchronous.** Synchronous checkpointing pauses training to save state, while asynchronous approaches overlap checkpointing with computation.
(2) **Local vs. remote storage.** Checkpoints may be written to fast local or remote storage, providing stronger fault tolerance.
(3) **Full vs. incremental.** Systems may store the entire training state or only the changes since the previous checkpoint.

2.2 Checkpointing Optimizations

Beyond these structural design choices, a significant body of research has explored mechanisms to reduce overheads associated with checkpointing and recovery in distributed training systems. The following approaches show several representative directions.

Overlapping checkpoint I/O with training. Several systems attempt to reduce checkpoint overhead by decoupling checkpoint construction from persistence and overlapping these operations with ongoing training. *CheckFreq* [14] introduces frequent, fine-grained checkpointing for deep neural networks by separating checkpoint creation into a snapshot phase (GPU→CPU copy) and a persistence phase (CPU→storage write), allowing the latter to overlap with training execution. *DeepFreeze* [16] extends this direction by integrating deep learning checkpointing with an asynchronous multi-level checkpointing runtime, enabling scalable checkpointing on large HPC systems. More recently, *DataStates-LLM* [13] has proposed "lazy asynchronous" checkpointing that exploits immutability windows of tensors during training to copy state in the background while minimizing interference with critical-path computation.

Reducing checkpoint size. Another line of work focuses on reducing the amount of data written during checkpointing. *Check-N-Run* [2] targets large-scale recommendation models and introduces differential checkpointing combined with quantization, saving only modified subsets of embeddings to significantly reduce storage and network bandwidth requirements. *IncrCP* [10] further refines incremental checkpointing by using dataflow tracing to track which embedding vectors are modified during training, enabling the system to reduce checkpoint construction and persistence overhead.

Leveraging memory hierarchy and replicas. Some systems exploit faster memory tiers or replicas to reduce checkpoint latency and improve recovery time. *Gemini* [24] proposes checkpointing large-model training state into CPU memory rather than directly to remote storage, leveraging higher memory bandwidth and introducing placement and scheduling mechanisms to balance recovery probability against training network interference. *PCcheck* [21] argues that checkpoint systems become bottlenecked by allowing only a single checkpoint pipeline in flight, and introduces support for multiple concurrent checkpoints that pipeline GPU→CPU staging and persistence more effectively.

2.3 Limitations in Evaluation Methodology

A common characteristic of the previously published checkpointing systems is that their evaluation relies heavily on real hardware deployments. These include production GPU clusters (*Check-N-Run*, *ByteCheckpoint*), public cloud GPU instances (*Gemini*, *PCcheck*), large HPC systems (*DeepFreeze*, *DataStates-LLM*), and multi-node GPU clusters with parallel file systems (*IncrCP*, *UCP*). Such environments provide highly realistic performance measurements, but they also impose practical limitations on experimentation.

First, access to such hardware is restricted to a small number of researchers. Second, the cost and operational complexity of running large-scale training experiments limit the number of configurations that can be explored. Finally, experiments on physical deployments make systematic *what-if* analysis difficult, such as studying checkpointing behaviour under different network topologies, storage placements, or bandwidth constraints.

2.4 Trace-Based Simulation for ML Systems

Synthetic workload traces have emerged as a practical approach for studying large-scale ML systems without requiring access to physical training clusters. The Chakra ETs [20] provide a standardised representation of distributed ML workloads, expressing operations as a directed acyclic graph (DAG) that can be replayed in simulation environments. At this level of abstraction, each DAG

node represents an operation such as compute or communication, while edges describe dependencies between operations. Compute nodes can include costs such as FLOPs or runtime estimates, and communication nodes describe collective operations together with metadata such as message size. Some recent work leverages traces collected from real deployments to drive simulation-based analysis of ML infrastructure. Frameworks such as MLSynth [19] and STAGE [12] extend this idea by enabling the generation of synthetic Chakra ETs directly from high-level model parameters and parallelisation strategies. MLSynth, in particular, generates one execution trace per distributed worker process, or rank, using an orchestrator to construct the baseline training operations and wrappers to extend traces with additional operations without modifying the core workload description. When coupled with compatible full-stack simulators such as ASTRA-sim2.0 [25], these traces allow end-to-end experimentation with ML workloads while maintaining high fidelity to realistic training behaviour. The extensible design of these frameworks creates an opportunity to embed additional system semantics, such as checkpointing, directly into trace generation, thereby enabling systematic exploration of checkpointing strategies within simulation.

3 CheckWork: Checkpoint-Aware Trace Generation Framework

To enable the trace-driven evaluation of checkpointing strategies, we design CheckWork, a new framework allowing the synthesis of checkpoint-aware Chakra ETs. The core design idea of CheckWork is to treat checkpointing operations as part of the execution graph of a training workload. This approach allows checkpoint activity to interact naturally with the structure of distributed training workloads, including synchronization points and communication phases. A significant advantage of adding such functionality at the trace generation level is that the rest of the tooling, such as the underlying simulator or hardware used to replay the workload, does not require any additional extensions or instrumentation.

The design of CheckWork is guided by the following goals aimed at enabling practical experimentation with checkpointing strategies in trace-driven ML workload analysis.

Expressiveness and configurability. The framework must support a wide range of checkpointing strategies used in distributed training systems. In practice, checkpoint mechanisms differ in synchronization semantics, persistence location, checkpoint frequency, checkpoint state size, and other parameters. CheckWork therefore aims to provide a configurable representation of checkpoint behaviour, allowing different strategies to be expressed and evaluated under identical workload conditions.

Compute & communication overhead modelling. Given that strategies vary in the nature of overhead introduced, it is important to model both the GPU-side and network-side effects. Operations such as model state snapshotting and local persistence are represented as compute overhead, while the transfer of checkpoint data across workers or to remote storage is modelled as communication overhead.

Compatibility with existing workflows. Another goal of the system is to integrate with existing ML infrastructure simulation workflows. CheckWork adheres to the Chakra ET schema [20], allowing generated traces to be replayed using existing simulators that support Chakra workloads. This ensures that checkpoint-aware traces can be analysed using the same tools that are already used to study distributed ML training.

Extensibility and usability. Finally, the framework is designed to support future checkpointing research and experimentation. The system is structured to allow new checkpointing strategies to be added easily without modifying the underlying workload model.

4 Implementation

We implement CheckWork as a trace-generation component that augments ML workload traces with checkpoint operations. We designed the current prototype to integrate with the MLSynth workload generator [19], which produces Chakra ETs from high-level model and parallelisation descriptions.

The resulting traces preserve the original training dependencies while incorporating additional nodes that model checkpoint behaviour. Because checkpointing is introduced entirely at trace generation time, the framework remains simulator-agnostic and compatible with Chakra-based simulators such as ASTRA-sim2.0 [25], Multiverse [6], and Mystique [9]. Figure 1 illustrates the resulting workflow.

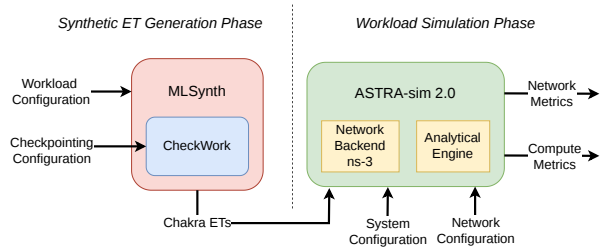

Figure 1: End-to-end pipeline for checkpoint-aware ML workload simulation. CheckWork generates synthetic Chakra traces augmented with checkpoint operations using MLSynth, which are simulated in ASTRA-sim2.0 to analyse checkpointing overhead.

CheckWork captures checkpointing at Chakra node granularity, so low-level effects such as tensor-level pipelining, pinned memory, RDMA, storage variability, and software-stack overheads are approximated through calibrated parameters rather than modelled as first-class resources.

4.1 Checkpoint Node Injection

During trace synthesis, the MLSynth orchestrator invokes the wrapper after the completion of each training iteration. Specifically, checkpoint operations are inserted after the backward pass and the data parallel synchronization step, in order to ensure that all workers have reached a consistent state before checkpointing begins. The wrapper then generates additional Chakra nodes representing checkpoint activity and appends them to the execution trace of each worker, or rank. Checkpoint operations reuse existing Chakra node types [20]. Local checkpointing is represented using compute-style nodes that model snapshot and persistence overhead, while remote checkpointing introduces communication nodes such as send and receive operations. Dependencies are added so that checkpoint operations interact correctly with the training workload and influence the execution critical path during simulation.

Table 1: Configuration parameters controlling checkpoint behaviour in CheckWork.

Parameter	Description
`mode`	Checkpoint strategy: `sync`, `async`, `remote_sync`, or `remote_async`.
`checkpoint_every_n`	Checkpoint frequency in iterations.
`state_multiplier`	Multiplier used to approximate full training state size (model + optimiser).
`overhead_multiplier`	Scales per-stage checkpoint compute cost and duration.
`kickoff_micros`	Duration of the checkpoint kickoff node (μs) in async and remote modes.
`max_inflight`	Cap on concurrent background writes; `1` serialises, `0` disables the cap.
`remote_target`	In remote modes, destination: `ring`, `rank0`, or `storage`.
`snapshot_multiplier` / `snapshot_micros`	Snapshot cost scale (bytes × multiplier) and optional duration override (μs).
`persist_multiplier` / `persist_micros`	Persist cost scale (bytes × multiplier) and optional duration override (μs).

We estimate the per-rank checkpoint volume from the size of the model state. Let P denote the total number of model parameters, b the number of bytes per parameter, and (t, p) the tensor- and pipeline-parallelism degrees. The per-rank checkpoint volume is

$$V = \frac{P}{t\,p}\, b\, \alpha\, s, \tag{1}$$

where α is the configurable `state_multiplier`, which can be set to account for additional checkpointed training state such as optimizer state, gradients, and metadata, and s is the MLSynth workload scaling factor used to scale the synthetic model size. Equation (1) estimates per-rank checkpoint volume in bytes and is used by CheckWork by default. The simulator derives the resulting training behavior based on the workload generated by CheckWork and network configurations including bandwidth and topology. In case the user prefers to use an alternative volume derivation model, the tool also allows overriding the default equation.

4.2 Configurability

Checkpoint behaviour in CheckWork is expressed through a configurable set of strategies that determine how and when checkpoint operations are executed during training. We summarize these parameters in Table 1. Rather than defining a fixed set of checkpoint implementations, we design this framework to expose a set of parameters that allow different strategies to be composed at trace generation time.

Synchronous vs. Asynchronous checkpointing. Synchronization semantics are expressed through dependencies in the execution trace. Synchronous configurations block the next iteration until checkpoint completion, while asynchronous configurations allow persistence operations to execute in the background. Figure 2 illustrates the differences in the produced traces for each mode.

Local vs. Remote checkpointing. Local checkpointing is represented using compute-style nodes that model the cost of writing checkpoint data to local storage. Remote checkpointing introduces additional communication nodes that transfer checkpoint state across the network, either through ring-based exchanges between workers or gather-style transfers to a designated rank or a storage persistence location, which is modelled as a virtual rank for simulator compatibility. These communication nodes use the same state-size model used for local checkpoints and are inserted into the Chakra ETs with appropriate dependencies to capture network overhead and interaction with training communication.

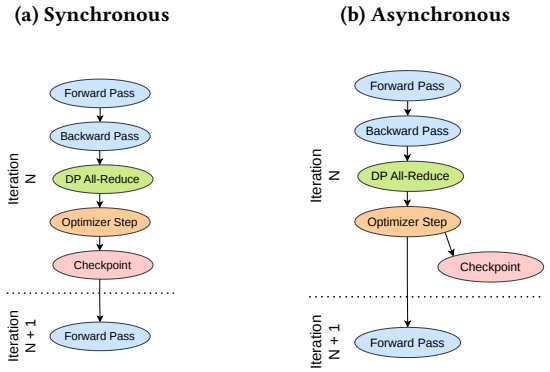

Figure 2: Chakra DAG representation of critical-path impact of synchronous and asynchronous checkpoint placement.

5 Evaluation

We evaluate CheckWork by reproducing results from prior checkpointing systems that were originally evaluated on real GPU test-beds. Specifically, Sections 5.1 and 5.2 demonstrate that the generated traces can faithfully model both GPU-side checkpoint overhead (CheckFreq [14]) and network-level interactions between training communication and checkpoint traffic (Gemini [24]).

5.1 Modelling Computational Overhead

To evaluate whether CheckWork can faithfully reproduce checkpointing behaviour observed in real systems, we model the checkpointing strategy proposed in CheckFreq [14]. CheckFreq reduces checkpointing overhead by performing a lightweight in-memory snapshot on the critical path while persisting the checkpoint to storage asynchronously in the background. As a result, only the snapshot operation delays the next training iteration, while the persistence phase is overlapped with subsequent computation.

Experiment Setup. We replicate the experimental configuration used in the CheckFreq evaluation on BERT training. The workload uses a BERT-like transformer model with 24 layers, hidden size 1024, and sequence length 512. Training runs for 200 iterations using data parallelism across 8 GPUs. We configure the workload using the checkpoint frequency reported in the original study and replicate the corresponding checkpoint size. All simulations are executed using the ASTRA-sim2.0 analytical engine [25], which estimates communication and checkpoint-transfer timing from the configured network topology and bandwidth rather than through packet-level simulation.

Workload Configuration. To mirror the evaluation presented in Figure 7 of the CheckFreq paper, we compare the checkpointing overhead relative to a baseline run without checkpointing. Specifically, we evaluate two configurations: (i) *synchronous checkpointing*, where both snapshot and persistence operations are on the critical path, and (ii) *CheckFreq-style checkpointing*, where only the

snapshot operation appears on the critical path while persistence executes in the background.

Results. Figure 3 compares the overhead reported in the CheckFreq paper with the results obtained using CheckWork. As expected, synchronous checkpointing introduces substantial slowdown due to blocking persistence operations, while the CheckFreq strategy reduces the overhead to only a small percentage of the baseline runtime. In our simulation, synchronous checkpointing results in approximately **71.2%** overhead, while CheckFreq-style checkpointing incurs only **2.7%** overhead. These results closely match the behaviour reported in the original study, where synchronous checkpointing introduces substantial slowdown while CheckFreq reduces overhead to only a few percent of the baseline runtime. Since the original evaluation was conducted on a real GPU test-bed (an 8-GPU DGX-1 system with NVIDIA V100 GPUs), this correspondence indicates that CheckWork can faithfully model GPU-side checkpointing overhead using generated ETs.

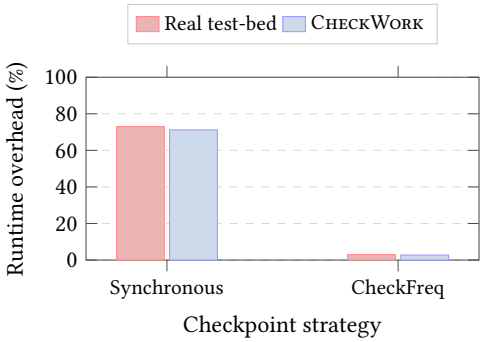

Figure 3: CheckWork successfully replicates the runtime overhead for BERT reported by CheckFreq [14].

Robustness study. In order to extend the existing evaluation of CheckFreq and validate the robustness of the qualitative results obtained via CheckWork, we also run a robustness study. As part of this, we simulate the same experiment at various checkpoint frequencies and sizes, and record the corresponding results for validation. Our conclusion holds true for various checkpoint frequencies and sizes, as expected from the CheckFreq paper. Figure 4 compares the full frequency/size sweep for CheckFreq and synchronous checkpointing, showing that CheckFreq performs better across all tested pairs in our experiment.

Cost Analysis. Thanks to trace-enabled analysis, the whole sweep ran in under 5 minutes on an Apple M5 laptop using the ASTRA-sim2.0 Analytical Engine, with peak observed memory usage of 1.5 GiB.

5.2 Modelling Network Interference

To further validate the ability of CheckWork to model checkpointing behaviour in distributed training environments, we replicate the checkpoint prioritisation mechanism proposed in Gemini [24]. Gemini addresses the problem of interference between training communication and checkpoint transfers when checkpoints are written to remote storage. The key idea is to prioritise training-related network communication over checkpoint traffic so that checkpoint transfers execute in the background without delaying critical training operations.

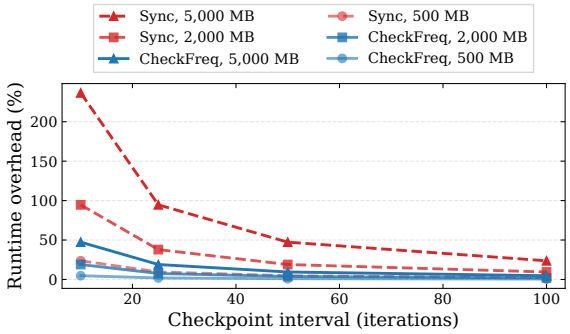

Figure 4: BERT runtime overhead under varying checkpoint sizes and cadences. CheckFreq consistently incurs lower overhead than synchronous checkpointing, especially for frequent checkpoints and larger checkpoint volumes.

Experiment Setup. We reproduce the distributed training configuration used in the Gemini evaluation. The workload simulates large-scale training across **32 GPUs** arranged in an oversubscribed three-tier fat-tree topology, using **3D parallelism** (tp=4, pp=2, dp=4) with a **1F1B pipeline schedule**. Tensor-parallel communication occurs within a node using a scale-up interconnect (*e.g.*, NVLink), while other communication occurs across the network. Checkpointing is performed asynchronously, generating background network traffic that competes with training communication. All simulations are executed using the ASTRA-sim2.0 ns-3 network backend [25]. We model prioritisation by placing training and checkpoint traffic in separate traffic classes, with checkpoint transfers assigned lower priority on shared links.

Workload Configurations. To mirror the evaluation presented in Figure 7 of the Gemini paper, we compare iteration time under two configurations: (i) a *baseline* run without checkpointing, and (ii) asynchronous checkpointing with *Gemini-style prioritisation*, where training communication is given higher network priority than checkpoint transfers.

Results. Figure 5 compares iteration time for the baseline and Gemini-style prioritisation. With asynchronous checkpointing and prioritised training traffic, iteration time increases only from **32.12 s** to **32.13 s**, corresponding to a negligible slowdown of **0.03%**. This closely matches Gemini's finding that prioritising training traffic prevents checkpoint transfers from affecting training throughput [24]. The agreement with results from Gemini's large-scale AWS GPU evaluation shows that CheckWork can capture network-level interference between checkpoint and training traffic in distributed training.

6 CheckWork Enables Further Analysis

We now use the generated ETs to support analyses that are difficult to run on physical test-beds, beyond just reproducing existing results. As an example, we focus on a claim in the Gemini paper [24] which argues that workload traffic should generally be prioritised over checkpointing traffic in single-job settings. CheckWork lets us test whether this insight extends to multi-job scenarios.

Observation. In large GPU clusters, multiple training jobs frequently share the same network fabric [8, 18]. In such multi-tenant environments, communication traffic from different jobs can interact in complex ways and lead to performance interference [1, 5, 26]. Using CheckWork, we analyse the execution traces of

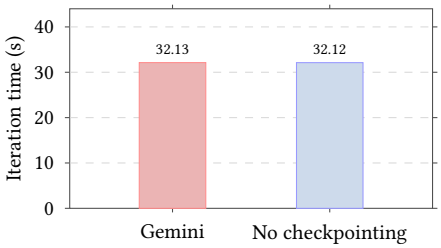

Figure 5: CheckWork enables simulating Gemini [24] strategy *without* requiring large-scale AWS clusters. Gemini-style prioritisation introduces a negligible overhead of 0.01 s per iteration (≈ 0.03%).

distributed training workloads and identify cases where communication phases across GPUs offer opportunities for optimisation. In particular, the nature of the pipeline scheduling results in some pipeline stages completing sooner than others in the drain stage. This means the final data parallel communication for these pipeline stages has *communication slack*, during which delaying a GPU's communication does not affect the overall iteration completion time. Figure 6 illustrates a simplified example with two jobs sharing the same network. Checkpoint transfers from one job can conflict with communication phases from another job, potentially causing interference. However, when communication slack exists, certain network transfers can be delayed or de-prioritised without affecting the critical path of the workload.

Implications. Gemini shows that prioritising training communication over checkpoint traffic effectively eliminates interference from asynchronous checkpoint transfers [24]. Our analysis suggests that the presence of communication slack opens additional opportunities for scheduling flexibility. In multi-job environments, checkpoint transfers could be scheduled more aggressively while selectively delaying non-critical communication phases from other jobs that fall within slack intervals. Such policies could reduce checkpointing time while preserving overall training throughput.

Experiment. To test our scheduling hypothesis, we use CheckWork to generate traces for two Transformer training jobs using synchronous remote checkpointing. Job 2 is configured to be 90% the size of Job 1, in order to recreate the exact scenario illustrated in Figure 6. We simulate the resulting traces using the ASTRA-sim2.0 ns-3 simulation engine on 32 GPUs configured as an oversubscribed

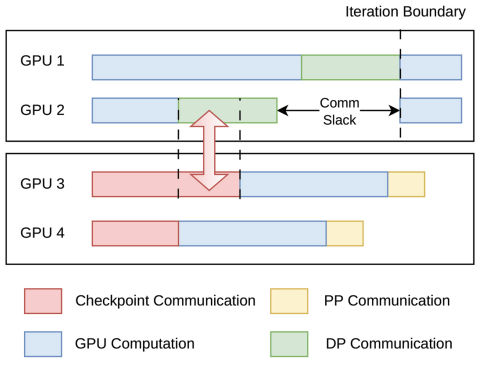

Figure 6: In multi-job settings, checkpoint traffic can interfere with other jobs' training communication.

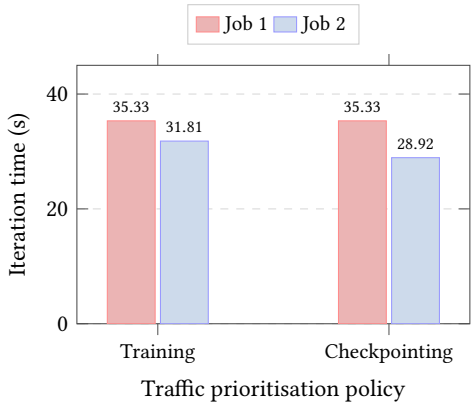

Figure 7: Prioritising checkpointing traffic reduces Job 2 iteration time without affecting Job 1, since checkpointing is prioritised over non-critical data-parallel communication.

three-tier fat-tree topology, such that traffic from the two jobs contends for shared network resources. The jobs are co-located within the same pod, with each job assigned 16 GPUs.

Results. Figure 7 reports the resulting iteration completion times, including checkpointing time. Prioritising checkpointing traffic reduces Job 2 iteration time by 9.1%, without affecting Job 1. We note that this improvement is specific to iterations where checkpoint traffic overlaps with non-critical data-parallel communication from another job. This result shows that checkpoint traffic can be prioritised over non-critical data-parallel communication when the latter is not on the critical path, contrasting with prior work, such as Gemini's approach [24]. More broadly, this shows how CheckWork enables trace-driven what-if analysis of checkpoint scheduling policies in multi-job training environments.

7 Conclusion

We presented CheckWork, a framework for generating checkpoint-aware execution traces for distributed ML training workloads. By augmenting Chakra ETs with checkpointing operations while preserving the original training dependencies, CheckWork enables checkpointing strategies to be expressed directly within workload traces and evaluated using existing ML system simulators. We believe CheckWork provides the missing piece to better understand the implications of checkpointing on the training job in unexplored scenarios, such as different topologies, various parallelisation strategies, and multi-job settings. We plan to extend CheckWork to (i) support additional checkpointing mechanisms and storage systems; (ii) integrate richer network and failure models for large-scale training clusters; and (iii) explore adaptive and network-aware checkpoint scheduling policies.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions on this paper. This work has been conducted with the support of the European Union EMPOWER-6G and FLECON-6G Doctoral Networks under Grant Agreement No. 101120332 and 101192462.

References

[1] Daiyaan Arfeen, Zhen Zhang, Xinwei Fu, Gregory Ganger, and Yida Wang. 2025. PipeFill: Using GPUs During Bubbles in Pipeline-parallel LLM Training. In *Eighth Conference on Machine Learning and Systems*. https://openreview.net/forum?id=650J0YFjnV

[2] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. 2022. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 929–943. https://www.usenix.org/conference/nsdi22/presentation/eisenman

[3] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.* 34, 3 (Sept. 2002), 375–408. doi:10.1145/568522.568525

[4] Aaron Grattafiori and et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[5] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 485–500. https://www.usenix.org/conference/nsdi19/presentation/gu

[6] Fei Gui, Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Ran Zhang, Hongbing Yang, and Dian Xiong. 2025. Accelerating Design Space Exploration for LLM Training Systems with Multi-experiment Parallel Simulation. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 473–488. https://www.usenix.org/conference/nsdi25/presentation/gui

[7] Tao He, Xue Li, Zhibin Wang, Kun Qian, Jingbo Xu, Wenyuan Yu, and Jingren Zhou. 2023. Unicron: Economizing Self-Healing LLM Training at Scale. arXiv:2401.00134 [cs.DC] https://arxiv.org/abs/2401.00134

[8] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 947–960. https://www.usenix.org/conference/atc19/presentation/jeon

[9] Mingyu Liang, Wenyin Fu, Louis Feng, Zhongyi Lin, Pavani Panakanti, Shengbao Zheng, Srinivas Sridharan, and Christina Delimitrou. 2023. Mystique: Enabling Accurate and Scalable Generation of Production AI Benchmarks. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 37, 13 pages. doi:10.1145/3579371.3589072

[10] Qingyin Lin, Jiangsu Du, Rui Li, Zhiguang Chen, Wenguang Chen, and Nong Xiao. 2024. IncrCP: Decomposing and Orchestrating Incremental Checkpoints for Effective Recommendation Model Training. *Proc. VLDB Endow.* 18, 4 (Dec. 2024), 1049–1062. doi:10.14778/3717755.3717765

[11] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark C. Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. 2020. CPR: Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. arXiv:2011.02999 [cs.LG] https://arxiv.org/abs/2011.02999

[12] Changhai Man, Joongun Park, Hanjiang Wu, Huan Xu, Srinivas Sridharan, and Tushar Krishna. 2025. STAGE: A Symbolic Tensor grAph GEnerator for distributed AI system co-design. arXiv:2511.10480 [cs.DC] https://arxiv.org/abs/2511.10480

[13] Avinash Maurya, Robert Underwood, M. Mustafa Rafique, Franck Cappello, and Bogdan Nicolae. 2024. DataStates-LLM: Lazy Asynchronous Checkpointing for Large Language Models. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing* (Pisa, Italy) *(HPDC '24)*. Association for Computing Machinery, New York, NY, USA, 227–239. doi:10.1145/3625549.3658685

[14] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 203–216. https://www.usenix.org/conference/fast21/presentation/mohan

[15] 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. 2021. 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* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. doi:10.1145/3458817.3476209

[16] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. 2020. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 172–181. doi:10.1109/CCGrid49817.2020.00-76

[17] OpenAI. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[18] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 1–18. https://www.usenix.org/conference/osdi21/presentation/qiao

[19] Adel Sefiane, Alireza Farshin, and Marios Kogias. 2025. MLSynth: Towards Synthetic ML Traces. In *Proceedings of the 2nd Workshop on Networks for AI Computing* (Coimbra, Portugal) *(NAIC '25)*. Association for Computing Machinery, New York, NY, USA, 98–104. doi:10.1145/3748273.3749211

[20] Srinivas Sridharan, Taekyung Heo, Louis Feng, Zhaodong Wang, Matt Bergeron, Wenyin Fu, Shengbao Zheng, Brian Coutinho, Saeed Rashidi, Changhai Man, and Tushar Krishna. 2023. Chakra: Advancing Performance Benchmarking and Co-design using Standardized Execution Traces. arXiv:2305.14516 [cs.LG] https://arxiv.org/abs/2305.14516

[21] Foteini Strati, Michal Friedman, and Ana Klimovic. 2025. PCcheck: Persistent Concurrent Checkpointing for ML. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 811–827. doi:10.1145/3669940.3707255

[22] 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. 2025. ByteCheckpoint: A Unified Checkpointing System for Large Foundation Model Development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 559–578. https://www.usenix.org/conference/nsdi25/presentation/wan-borui

[23] Xizheng Wang, Qingxu Li, Yichi Xu, Gang Lu, Dan Li, Li Chen, Heyang Zhou, Linkang Zheng, Sen Zhang, Yikai Zhu, Yang Liu, Pengcheng Zhang, Kun Qian, Kunling He, Jiaqi Gao, Ennan Zhai, Dennis Cai, and Binzhang Fu. 2025. SimAI: Unifying Architecture Design and Performance Tuning for Large-Scale Large Language Model Training with Scalability and Precision. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 541–558. https://www.usenix.org/conference/nsdi25/presentation/wang-xizheng-simai

[24] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. 2023. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 364–381. doi:10.1145/3600006.3613145

[25] William Won, Taekyung Heo, Saeed Rashidi, Srinivas Sridharan, Sudarshan Srinivasan, and Tushar Krishna. 2023. ASTRA-sim2.0: Modeling Hierarchical Networks and Disaggregated Systems for Large-model Training at Scale. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 283–294. doi:10.1109/ISPASS57527.2023.00035

[26] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao