wdCP: Windowed Incremental Checkpointing for Efficient and Bounded LLM Recovery

Wendi Cheng
Northwestern Polytechnical University
Xi'an, China
chengwendi@mail.nwpu.edu.cn

Xiao Zhang*
Northwestern Polytechnical University
Xian, China
zhangxiao@nwpu.edu.cn

Xiaonan Zhao
Northwestern Polytechnical University
Xi'an, China
zhaoxn@nwpu.edu.cn

Xiaoling Shu
Northwestern Polytechnical University
Xi'an, China
shuxiaoling@mail.nwpu.edu.cn

Jinjiang Wang
Northwestern Polytechnical University
Xi'an, China
jinjiangwang@mail.nwpu.edu.cn

Shujie Han
Northwestern Polytechnical University
Xi'an, China
shujiehan@nwpu.edu.cn

Abstract

Checkpointing is essential for fault tolerance in large-scale LLM training, yet periodic full-state checkpoints bring heavy I/O overhead and training stalls. Prior work suggests that differential checkpointing is ineffective for LLMs, since most parameters updates every iteration, leading to dense updates. This paper revisit this assumption and observe that parameter updates are naturally generated inside optimizer execution and exhibit significant temporal and layer-wise heterogeneity. Guided by this, we present wdCP, a lightweight runtime that captures optimizer-level parameter deltas and asynchronously persists them using a windowed buffering mechanism. wdCP further introduces lightweight anchor snapshots to bound recovery cost. We implement wdCP and evaluate it on several representative models. Results show that wdCP introduces less than 5% training overhead while achieving up to 69.2× reduction in checkpoint size and enabling fast, bounded recovery.

CCS Concepts

• **Software and its engineering → Checkpoint/Restart**; • **Information systems → Data management systems**.

Keywords

LLM Training, Incremental Checkpointing, Fault Tolerance

ACM Reference Format:
Wendi Cheng, Xiao Zhang, Xiaonan Zhao, Xiaoling Shu, Jinjiang Wang, and Shujie Han. 2026. wdCP: Windowed Incremental Checkpointing for Efficient and Bounded LLM Recovery. In *Proceedings of the 23rd ACM International Conference on Computing Frontiers (CF '26), May 19–21, 2026, Catania, Italy*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3801487.3806069

*Corresponding author: zhangxiao@nwpu.edu.cn

CF '26, Catania, Italy

ACM ISBN 979-8-4007-2568-5/2026/05
https://doi.org/10.1145/3801487.3806069

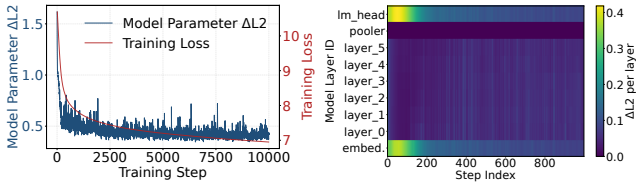
1 Introduction

Checkpointing is a fundamental mechanism to ensure fault tolerance in large language model (LLM) training [5]. Modern training jobs often run for days or weeks in GPU clusters, making periodic checkpoints necessary to preserve progress under failures or preemptions [7]. In practice, large-scale LLM training systems adopt frequent checkpointing (e.g., every few hundred steps) to limit recovery loss [10]. However, as model sizes and training scales grow, checkpointing has become a bottleneck due to the cost of repeatedly persisting full model and optimizer states to storage [3, 11]. To reduce this overhead, prior works explore incremental or differential checkpointing on recommendation models, which stores only parameter updates instead of full checkpoint [2, 6]. In contrast, LLM training updates most parameters every iteration, making deltas appear dense and leading to the common belief that differential checkpointing is ineffective for LLM training [12].

We find that this assumption is overly conservative. Although LLM updates are dense, they exhibit significant redundancy and heterogeneity across layers and training phases. Recent work such as ExCP [4] shows that optimizer states and parameter updates can be heavily compressed, suggesting that delta storage is not inherently prohibitive. Our analysis further shows that parameter update magnitudes vary significantly over time and across layers, indicating opportunities for efficient delta-based checkpointing.

Motivated by these, we present wdCP, a runtime system for incremental checkpointing in LLM training. wdCP captures optimizer-level parameter deltas during optimizer execution, buffers them in a windowed pipeline, then compresses and persists them asynchronously without blocking the training critical path. To ensure efficient recovery, wdCP further introduces lightweight anchor snapshots that bound the length of delta replay. By exploiting the structure of optimizer updates and training dynamics, wdCP enables practical differential checkpointing for LLM workloads with significantly reduced storage overhead. In summary, wdCP makes the following contributions.

- We propose an incremental checkpoint runtime for LLM training that replaces full-state storage with optimizer-level delta tracking, significantly reducing checkpoint size while preserving exact recovery semantics.



(a) Parameter change (L2 norm) (b) Layer-wise update heatmap

Figure 1: Parameter update dynamics during BERT training. (a) Global magnitude. (b) Layer-wise heterogeneity.

- We design a training-integrated delta capture and window-based lazy persistence mechanism that decouples checkpointing from training and enables frequent checkpoints.
- We introduce an anchor-based bounded recovery scheme that avoids loading optimizer states and limits replay length, achieving fast and predictable recovery.

2 Motivation and Observations

Incremental checkpointing is widely explored to optimize recommend models checkpoints, but applying it to LLM training remains challenging [12]. First, existing systems typically compute deltas by explicitly computing model states at checkpoint time, introducing additional computation, synchronization, and device transfer overhead. Second, although compression can significantly reduce delta size, many approaches rely on compute-intensive compression routines that are difficult to integrate into the training critical path. Finally, differential checkpointing shifts overhead from checkpoint creation to recovery, where reconstructing the model requires replaying long chains of updates, potentially leading to high recovery latency. We analyze parameter update characteristics during LLM training. Results from BERT trainning (Figure 1a and 1b) show that update magnitudes vary significantly across both training progress and model layers. Early training produces larger updates, while later stages become more stable, and middle Transformer layers tend to change more than embeddings or normalization layers.

3 Design

wDCP follows three principles: leveraging training semantics to avoid explicit parameter difference computation; decoupling delta capture from persistence to prevent blocking the training critical path; and organizing checkpoints to ensure bounded recovery cost.

3.1 Design Overview

Figure 2 illustrates the overview of wDCP runtime. The key insight of wDCP is to model training as a sequence of optimizer-induced state transitions and to capture and persist these transitions through a decoupled pipeline outside the training critical path.

First, wDCP performs *low-overhead delta capture* within *optimizer.step()*. Instead of computing state differences after the update, wDCP intercepts the parameter deltas that are already produced by the optimizer's computation path. These deltas are immediately transferred to pinned host memory using asynchronous device-to-host copies issued on the optimizer CUDA stream. Because this transfer fully overlaps with next forward and backward computation, it introduces negligible overhead to training. Second, wDCP

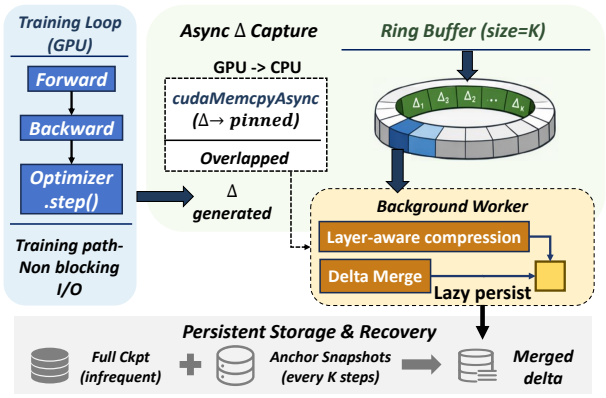


Figure 2: overview of wDCP runtime.

adopts *lazy delta persistence* through a bounded *Ring Buffer*. Rather than persisting deltas at every step, recent deltas are accumulated in a fixed-size pinned-memory window. This windowed buffering defers delta processing to a background stage, where layer-aware compression and consolidation can be performed based on statistics observed across multiple steps. Finally, wDCP ensures *bounded recovery* using lightweight anchors. Periodic anchor snapshots subdivide long delta chains into short segments, which guarantees that recovery latency is bounded by a fixed small size.

3.2 Low-overhead Delta Capture and Processing

A key observation is that parameter updates are directly computed within the optimizer and do not require explicit computing. In PyTorch optimizer implementations such as Adam/AdamW, especially in the default `_multi_tensor_adam` (foreach) execution path, the parameter update amount $\Delta\theta_t$ is explicitly computed as part of the optimizer's computation flow before actually overwriting model parameters. During batched tensor updates, the increments are materialized as tensor lists in GPU memory. Rather than deriving them through delayed $\theta_{t+1}-\theta_t$ differencing, wDCP captures them at generation time, reorganizes them into a contiguous layout, and integrates them into the parameter write-back path. Operating within the optimizer computation path, it avoids extra kernel launches, memory scans, and parameter comparisons, incurring negligible overhead on the training critical path.

In addition, wDCP adopts a producer–consumer design, where GPU-generated parameter deltas act as the producer and CPU workers serve as consumers responsible for post-processing. After each `optimizer.step()`, deltas are asynchronously transferred from device to host via non-blocking device-to-host copies issued on a dedicated copy stream, and are placed into a pinned *Ring Buffer* in host memory. CUDA events are used to signal transfer completion, enabling background CPU workers to safely access, compress, and persist deltas without blocking the training thread. This design allows delta transfer, compression, and persistence to overlap with ongoing training computation, ensuring that checkpoint-related operations remain off the training critical path.

3.3 Lazy Delta Compression

To efficiently persist the buffered delta data, it must eventually be persist. However, directly persisting every full captured delta incurs

high I/O overhead and stalls training. To avoid this, wdCP adopts a *window-based lazy persistence strategy*, where step-level deltas are first accumulated in CPU memory, and only consolidated data are written to disk with fine-grained compression.

The design is rooted in Observation 2, which exposes two challenges for delta-based checkpointing. First, due to strong phase-dependent training dynamics, the same layer can exhibit drastically different update magnitudes at different stages of training. As a result, the importance of a delta cannot be reliably determined at any single step. A per-step delta is often too small and noisy to indicate which updates are meaningful, making compression or pruning decisions at this level unreliable. This temporal uncertainty necessitates the use of a multi-step observation window as the proper unit of analysis. Second, there exists severe spatial heterogeneity in the information content of deltas across layers. The middle Transformer layers consistently carry high update energy, while embedding and normalization layers change very little. Treating all parameter deltas equally for persistence or compression therefore leads to substantial redundancy and inefficiency.

To enable this, wdCP introduces a *Ring Buffer* that maintains a fixed memory window of recent deltas, deferring delta processing to the background stage and enabling layer-aware compression, merging, and persistence. The host-memory overhead of the *Ring Buffer* is bounded and explicitly controlled by the system configuration. Specifically, the per-node memory footprint is upper-bounded by the number of GPU workers, the window size K, the local parameter shard count, and the storage size of each parameter delta. In distributed training frameworks such as FSDP and ZeRO, parameters and optimizer states are sharded across GPU workers, so the buffer footprint scales with the local shard size rather than the full model size. As a result, the memory overhead remains modest and fits within the DRAM capacity of modern training nodes (e.g., DGX A100, H100 with 1–2 TB memory). Consequently, wdCP trades a small amount of additional host memory for improved robustness and reliability. In the rare case of sustained throughput mismatch that fills the buffer, wdCP triggers a fallback mechanism that performs a synchronous checkpoint to flush buffered state, resets the buffer, and resumes normal execution.

wdCP uses a *Ring Buffer*-based pipeline to manage delta capture and persistence. At initialization, wdCP pre-allocates a fixed-size *Ring Buffer* in pinned host memory, where each slot stores the delta from one training step. The buffer is maintained per worker, allowing each GPU to interact only with its corresponding CPU thread without cross-worker contention. After a delta Δ_t is generated in *optimizer.step()*, it is transferred to the next slot via a non-blocking device-to-host copy, overlapping with next training step. A background CPU worker monitors CUDA events to detect transfer completion. Once ready, it applies lightweight compression to the delta and prepares it for persistence. We adopt a compression scheme similar to ExCP [4], which exploits redundancy across consecutive updates through residual encoding and quantization.

3.4 Anchor-Based Bounded Recovery

A fundamental challenge of differential checkpointing is that recovery cost grows with the length of the delta chain. Let the full checkpoint be taken at step t_0, and let the failure occur at step t_f.

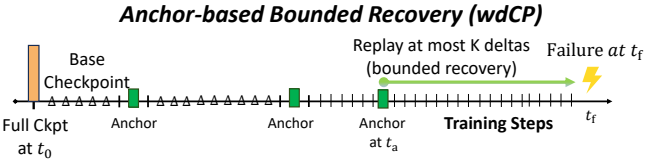


Figure 3: Anchor-based bounded recovery in wdCP.

As training progresses, $t_f - t_0$ can become large, causing recovery latency to grow linearly with training time.

The key observation is that recovery overhead is dominated not by the size of deltas, but by the *length of the replay chain*. wdCP reduces recovery cost by bounding replay length via *lightweight anchors* inserted every K steps. The parameter K introduces a trade-off between recovery cost and checkpoint efficiency. A larger K enables better aggregation but increases replay cost, while a smaller K reduces replay cost at the expense of aggregation. In practice, we treat K as a fixed system parameter and find that moderate values provide stable performance across training phases.

An anchor is a snapshot of model parameters at step t_a, serving as an intermediate recovery point, i.e., $A_{t_a} = \theta_{t_a}$. Unlike a full checkpoint, an anchor does not include optimizer states. In wdCP, each training step logs semantic optimizer deltas in the form $(\Delta\theta, \Delta m, \Delta v)$ after `optimizer.step()`. These deltas fully capture the evolution of both model parameters and optimizer moments. Therefore, persisting optimizer states in anchors is unnecessary, allowing anchors to remain small while preserving exact recovery semantics. With anchors commited periodically, recovery starts from the nearest anchor rather than the base snapshot, requiring replay of at most K deltas. As shown in Figure 3, suppose a failure occurs at step t_f, and the most recent anchor was taken at step t_a such that $t_f - t_a \leq K$. Recovery proceeds from the anchor by initializing the state as $(\theta_{t_a}, 0, 0)$ and sequentially replaying the logged deltas: $S_{t_f} = (\theta_{t_a}, 0, 0) + \sum_{i=t_a+1}^{t_f} (\Delta\theta_i, \Delta m_i, \Delta v_i)$. Here, $(\Delta\theta_i, \Delta m_i, \Delta v_i)$ denote the per-step updates to the model parameters and the first- and second-order moments at step i, respectively. wdCP employs a streaming pipeline that overlaps storage reads, CPU decompression, and state reconstruction, effectively hiding decompression and preventing it from becoming a bottleneck.

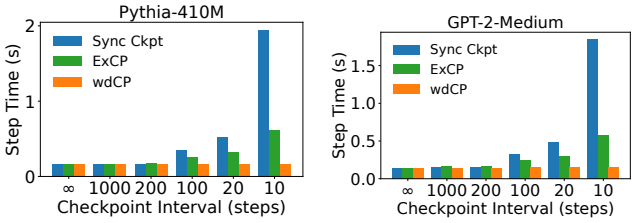
4 Evaluation

Setup. We evaluate wdCP on its ability to minimize training stalls, reduce persistence volume, and provide fast, bounded recovery. Experiments are conducted on a GPU server (Ubuntu 20.04, 2× Xeon Silver 4210, 256 GB RAM, 4× V100-16GB, CUDA 12.4). Models are implemented in Python 3.10 and PyTorch 2.9.

Baseline and Models. We compare wdCP with IncrCP [6] and ExCP [4]. We evaluate wdCP on DLRM [8], Pythia-410M [1], GPT-2-Medium [9] and TinyLlama-1.1B [13], covering both recommendation and dense LLM workloads. All models use the `AdamW` optimizer with default datasets.

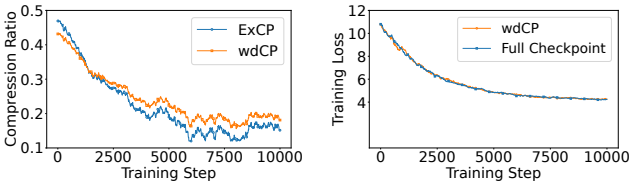
4.1 Experiment Results

4.1.1 Training Overhead. We first assess whether wdCP affects per-step training time. For DLRM, synchronous checkpointing introduces per-step overhead (62.487,s) compared to the no-checkpoint



(a) Pythia-410M **(b) GPT-2-Medium**

Figure 4: Impact of checkpoint frequency on training step time across different models.



(a) Compression ratio. **(b) Training loss.**

Figure 5: Compression effectiveness and training accuracy of wdCP during 10,000 steps of Pythia-410M training.

ideal baseline (0.049,s), while incremental methods and wdCP drastically reduce this cost. Specifically, wdCP achieves 0.178,s per step, corresponding to a 2.2× speedup over IncrCP via optimizer-integrated delta capture. On Transformer-based LLMs (Pythia, GPT-2, TinyLlama), wdCP incurs only 2.9%–4.0% overhead relative to the no-checkpoint baseline, with stable per-step time across models.

We next investigate how checkpoint interval affects training performance, evaluating various models with intervals from 10 to 1000 steps (∞ for No Checkpointing). The checkpoint interval denotes logical checkpoints where the training state must be recoverable: baselines materialize full checkpoints at each interval, while wdCP provides the same recoverability guaranty, which commits all prior updates at step N to enable failure recovery to that step. As shown in Figure 4, synchronous checkpointing incurs severe slowdown at high frequencies on Transformer models: on Pythia-410M, per-step time increases from 0.162 s to 1.95 s (12× slowdown), and on GPT-2-Medium from 0.145,s to 1.85,s (12.7×). In contrast, wdCP maintains nearly constant step time across all checkpoint intervals.

4.1.2 Checkpoint Size Reduction and Training Accuracy. We train Pythia-410M for 10,000 steps and compare checkpoint size and training accuracy under wdCP and ExCP. Both the checkpoint interval and window size are set to 10. wdCP and ExCP exhibit similar compression behavior. As shown in Figure 5, during early training, the compression ratio ranges from 35%–45%, and gradually decreases to 12%–18% after 6,000 steps as redundancy increases. While achieving similar compression effectiveness, ExCP compresses full checkpoints synchronously, whereas wdCP performs window-based delta compression asynchronously. This allows wdCP to achieve comparable compression without interfering with the training critical path. The training loss of wdCP (subfigure b) closely matches PyTorch across all 10,000 steps, showing no observable deviation in convergence.

4.1.3 Bounded Recovery with Anchor Snapshots. We evaluate recovery by injecting failures during Pythia-410M training and measuring the reovery time with a window size of K=10. Compared to conventional `torch.load()`, wdCP reduces recovery data from 4.53 GB to 67 MB (69.2× smaller) and shortens recovery time from 8.17 s to 1.12 s (7.3× faster). Without anchors, recovery time increases with training progress due to replaying all accumulated deltas. With anchors inserted every K=10 steps, recovery time remains nearly constant, as at most K deltas are replayed.

5 Conclusion

This paper presents wdCP, an incremental checkpoint runtime for LLM training. By capturing optimizer-level deltas and decoupling persistence from the training path, wdCP enables low overhead checkpointing. A window-based lazy persistence mechanism reduces storage, while anchor snapshots bound recovery cost. Experiments show that wdCP maintains stable training performance, significantly reduces checkpoint size, and for fast bounded recovery. In future work, we plan to extend wdCP to distributed training frameworks by integrating with data-parallel and model-parallel mechanisms (e.g., DeepSpeed ZeRO) to support sharded state management and cross-node coordinated recovery capabilities.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 62502391 and No. 62272394).

References

[1] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, and Herbie Bradley. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *Proc. of ICML*. 2397–2430.
[2] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, and Dheevatsa Mudigere. 2022. Check-N-Run: A checkpointing system for training deep learning recommendation models. In *Proc. of USENIX NSDI*. 929–943.
[3] Hongliang Li, Zichen Wang, Hairui Zhao, and Meng Zhang. 2025. Convergence-aware optimal checkpointing for exploratory deep learning training jobs. *Future Generation Computer Systems* (2025).
[4] Wenshuo Li, Xinghao Chen, Han Shu, and Yehui Tang. 2024. ExCP: Extreme LLM checkpoint compression via weight-momentum joint shrinking. In *Proc. of ICML*. 27575–27588.
[5] Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, and Masahiro Tanaka. 2025. Universal checkpointing: A flexible and efficient distributed checkpointing system for large-scale DNN training with reconfigurable parallelism. In *Proc. of USENIX ATC*. 1519–1534.
[6] Qingyin Lin, Jiangsu Du, Rui Li, and Zhiguang Chen. 2024. IncrCP: Decomposing and orchestrating incremental checkpoints for effective recommendation model training. *Proc. of VLDB Endow.* 18, 4 (2024), 1049–1062.
[7] Avinash Maurya, Robert Underwood, M. Mustafa Rafique, and Franck Cappello. 2024. Datastates-LLM: Lazy asynchronous checkpointing for large language models. In *Proc. of ACM HPDC*. 227–239.
[8] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, and Huang. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).
[9] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.
[10] Borui Wan, Mingji Han, Yiyao Sheng, and Yanghua Peng. 2025. ByteCheckpoint: A unified checkpointing system for large foundation model development. In *Proc. of USENIX NSDI*. 559–578.
[11] Guanhua Wang, Olatunji Ruwase, Bing Xie, and Yuxiong He. 2024. *FastPersist: Accelerating Model Checkpointing in Deep Learning*. Technical Report. Microsoft Research. https://arxiv.org/abs/2406.13768 Preprint.
[12] Zhiqiang Wang, Wenzhe Zhu, Zaigui Zhang, and Chaomei Yan. 2025. Amber: Towards fast and space-efficient incremental checkpointing in large language model training. In *Proc. of ICPP*. 678–688.
[13] Peiyuan Zhang, Guangtao Zeng, Tianduo Wang, and Wei Lu. 2024. TinyLlama: An Open-Source Small Language Model. arXiv:2401.02385 [cs.CL]