Mnemosyne: Lightweight and Fast Error Recovery for LLM Training in a Just-In-Time Manner

Jinyi Xia
Beihang University
Beijing, China
jinyi.xia@outlook.com

Menghao Zhang
Beihang University
Beijing, China
zhangmenghao@buaa.edu.cn

Jiaxun Huang
Beihang University
Beijing, China
huangjx@buaa.edu.cn

Yuezheng Liu
Beihang University
Beijing, China
liuyuezheng@buaa.edu.cn

Xiaohe Hu
Infrawaves
Beijing, China
huxiaohe@infrawaves.com

Xudong Liu
Beihang University
Beijing, China
liuxd@buaa.edu.cn

Chunming Hu
Beihang University
Beijing, China
hucm@buaa.edu.cn

Abstract

With the rapid scaling of large language model (LLM) training clusters, GPU errors frequently occur and disrupt the training process. While traditional error recovery methods, such as periodic checkpointing, are effective, they incur substantial overhead in both daily operations and recovery processes. Just-in-time checkpointing, a representative alternative, reduces this overhead by eliminating the periodic checkpoint saving procedure and optimizing the recovery workflow. However, its complex GPU context decoupling mechanism and global reinitialization of communication backend remain resource-intensive and slow. In this paper, we present Mnemosyne, a lightweight and fast error recovery framework for LLM training. To minimize both daily and recovery overhead, Mnemosyne optimizes the GPU context decoupling component with shared-memory-based IPC and index-based handle mapping, and designs a flexible collective communication library that dynamically adjusts links of built communicators without requiring reinitialization. Preliminary experiments on our open-source prototype demonstrate that, compared to the state of the art, Mnemosyne reduces daily overhead by up to 58.8% and communication rebuilding time by up to 91.3%.

CCS Concepts

• Computer systems organization → Reliability; **• Computing methodologies → Machine learning**; **Distributed computing methodologies**; **• Networks → Data center networks**.

APNET 2025, Shang Hai, China

ACM ISBN 979-8-4007-1401-6/25/08
https://doi.org/10.1145/3735358.3735372

ACM Reference Format:
Jinyi Xia, Menghao Zhang, Jiaxun Huang, Yuezheng Liu, Xiaohe Hu, Xudong Liu, Chunming Hu. 2025. Mnemosyne: Lightweight and Fast Error Recovery for LLM Training in a Just-In-Time Manner. In *9th Asia-Pacific Workshop on Networking (APNET 2025), August 07–08, 2025, Shang Hai, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3735358.3735372

1 Introduction

Large Language Models (LLMs) such as OpenAI o1 [25] and DeepSeek-R1 [10] have achieved great success in a spectrum of tasks, and their generality and performance continue to improve with the model size and the training data increasing [16]. For example, GPT-2 [27] was released with 1.5 billion parameters in 2019, while three years later, PaLM [4] was shown with 540 billion parameters. As the model size and the training data grows, LLM training requires more hardware, making training failures more frequent. For example, Meta used 992 NVIDIA A100s to train OPT-175B, and encountered about 110 failures in two months, causing the waste of 178,000 GPU hours [36]. All these factors bring significant challenges to the efficiency of model training, and also raise higher requirements for fault tolerance.

Traditional fault tolerance predominantly relies on periodic checkpointing (e.g., DeepFreeze [23], CheckFreq [22], Check-N-Run [7], Gemini [35], etc.), where model states are saved at fixed intervals (e.g., every N iterations) for recovery. Although this method provides basic recovery capabilities, it introduces significant overheads during both training and recovery phases. Frequent checkpointing slows down the training progress due to the time spent on serializing large model states. Meanwhile, recovery requires reloading checkpoints and replaying iterations since the last save, leading to substantial recomputation. These limitations become prohibitive as model sizes and cluster scales grow.

Just-in-time checkpointing [11] emerges as a promising alternative by leveraging data parallelism (DP) replicas to enable rapid recovery. This approach minimizes steady-state overhead and restricts iteration recomputation to at most one training step. Despite these benefits, existing designs face two primary limitations.

First, their reliance on complex device proxies, which are ported directly from Singularity [31] and initially designed for elasticity, introduces unnecessary overhead during training process. Second, they only retain computation resources (e.g., GPU buffers) but neglect communication resources during recovery, requiring global reconstruction of communication backends, e.g., NCCL [24], which consumes considerable time during recovery (§2).

To address these challenges, we present Mnemosyne, a fault tolerance framework that preserves the benefits of just-in-time checkpointing while addressing its limitations. Mnemosyne first introduces a lightweight device proxy architecture optimized for fault recovery rather than general elasticity, reducing steady-state operational costs. Furthermore, Mnemosyne designs a flexible collective communication library (CCL) that supports runtime link adjustment without full reinitialization, enabling partial communication topology reconstruction localized to failed nodes, and bypassing global coordination overhead.

The technical foundation of Mnemosyne lies in two key techniques. First, the lightweight device proxy employs call interception to isolate failures and transparently migrate tasks across nodes. Its shared-memory-based IPC and nearly zero-overhead handle mapping bring efficiency to the decoupling of logical tasks from physical hardware, ensuring minimal interference during normal operations while facilitating rapid recovery. Second, the flexible CCL adopts a metadata-driven approach for communication initialization. Instead of globally renegotiating connections during recovery, the framework dynamically updates specific links using predefined topology templates. This selective adjustment eliminates the need for full reinitialization and enables runtime node replacement, addition, and removal via dynamic link adjustment. For validation and evaluation, we implement an open-source prototype of Mnemosyne [2] with the basic device proxy and flexible CCL. Preliminary experiments demonstrate that Mnemosyne brings only an average of 3.6% daily overhead during training process, which is reduced by up to 58.8% compared with Just-in-time checkpointing [11]. Besides, the communication reinitialization time is also reduced by up to 91.3% compared with original NCCL [24]. We hope Mnemosyne can inspire truly lightweight and fast fault tolerance mechanisms for next-generation LLM training frameworks.

2 Background and Motivation

2.1 LLM Training Fault Tolerance

Today's LLM training system is faced with frequent and various errors due to the increasing hardware scale, and most training errors are related to GPU. According to DeepSeek's records [1], GPU-related errors include software-caused errors, NVLink errors, memory ECC errors, uncorrectable GPU failures, etc. Among them, NVLink errors are the most common one, accounting for 42.57% of the total [1]. Shanghai AI Lab [12] also reports that GPU-related errors consume more than 66% of GPU time in a cluster of 2,000+ GPUs, wasting over 2,400,000 GPU minutes in 6 months. Despite various causes, all these GPU errors can be categorized into two types: recoverable ones and irrecoverable ones. Recoverable errors, e.g., NVLink errors and memory ECC errors, can be solved by mere restart, while irrecoverable errors, e.g., GPU system processor errors, are usually caused by hardware faults and may need manual fixing or even return material authorizations. According to DeepSeek, they only encountered 1 irrecoverable error among past year's GPU-related errors, with a proportion of less than 0.01% [1]. Conclusively, LLM training errors are costly, common, but mostly recoverable.

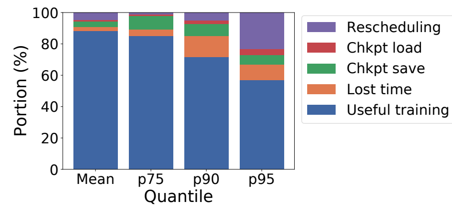


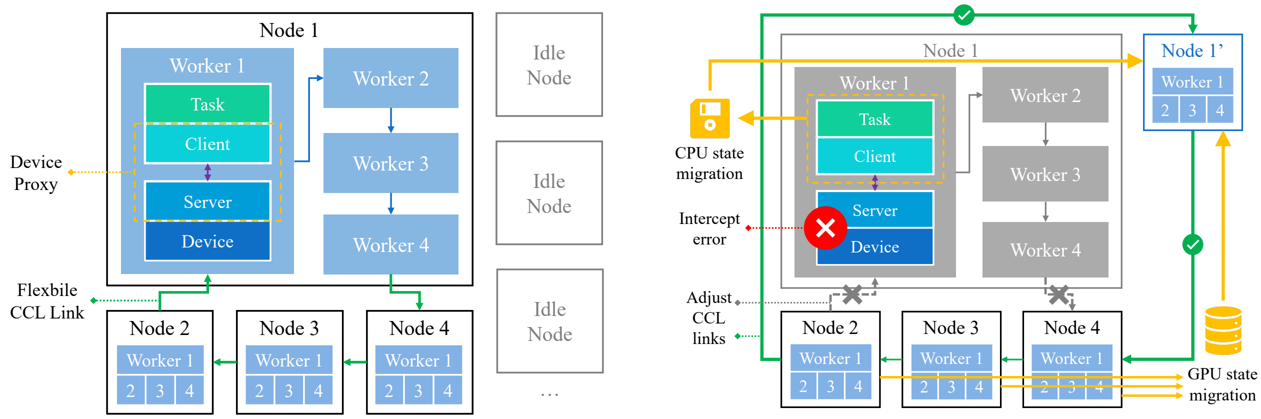
Figure 1: The portion of periodic-checkpoint-related overheads by Facebook [19].

The mainstream solution to error recovery for LLM training is periodic checkpointing, e.g., DeepFreeze [23], CheckFreq [22], Check-N-Run [7], Gemini [35], MegaScale [15], etc., all of them sharing the same routine. During the training process, the model parameters and optimizer states are saved every several iterations. Once a failure occurs, the training task is restarted and the last checkpoint is loaded. This method is suitable for all kinds of deep learning training tasks, but it may consume plenty of extra computation resources: (1) saving checkpoints interferes with the training progress, (2) restarting the task and loading the checkpoint take considerable time, and (3) the lost iterations need to be redone. Beside the overhead from the periodic checkpointing mechanism itself, the cluster scheduler needs to find available nodes as alternatives for the failed nodes, bringing extra rescheduling overhead. According to statistics from Facebook [19], periodic checkpointing leads to a 43% training time drop at most (Figure 1). Therefore, periodic checkpointing is effective but not efficient enough.

With the development of LLMs, error recovery methods with higher efficiency are designed based on their unique properties. LLMs apply multiple parallelism strategies for efficient training [37], such as data parallelism (DP), tensor parallelism, pipeline parallelism, expert parallelism, etc. Pioneering work, i.e., just-in-time checkpointing [11], designs mechanisms using LLM's DP mechanism to achieve more efficient checkpointing. It does not need to save checkpoint periodically in steady state, avoiding interference with the training progress. Once an error occurs, only the restarted nodes need loading checkpoints from the DP replicas, avoiding loading checkpoints globally, so its recovery is greatly more lightweight and swift than periodic checkpointing.

2.2 Observation and Motivation

Despite its advantages, just-in-time checkpointing is far from perfection. Some of its disadvantages still bring significant efficiency degradation to LLM training jobs. First, for overhead in steady-state work, although it avoids checkpointing periodically, its mechanism still introduces extra daily overhead. To provide users with transparent just-in-time checkpointing, the framework needs to intercept error, recover the state for error nodes, and log and replay training frameworks' operations. Device proxy, a key component to take on

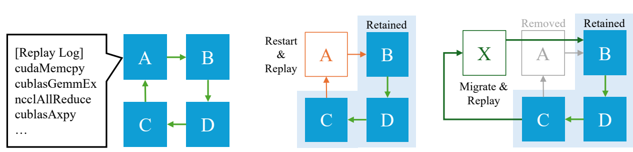


(a) Steady state. The device proxy and flexible CCL are marked. **(b) Recovery state. The key steps of the workflow are marked.**

Figure 2: Framework of Mnemosyne.

Table 1: Time (seconds) taken for each step of error recovery in just-in-time checkpointing by Microsoft [11]. Bert-B-FT and GPT2-S-3D are evaluated with 8×NVIDIA V100s. GPT2-S and PyramidNet are evaluated with 4×NVIDIA A100s.

Step	Bert-B-FT	GPT2-S	GPT2-S-3D	PyramidNet
Delete communicators and GPU handles	1.013	0.779	0.831	0.850
Recreate NCCL communicators	1.054	8.340	15.54	1.038
Reset GPU buffers	0.001	0.001	0.001	0.002
Recreate GPU handles	0.006	0.004	0.004	0.027
Replay minibatch APIs	0.006	0.004	0.002	0.004



(a) Steady-State Operation. **(b) Recovery in place.** **(c) Recovery with Migration.**

Figure 3: Workflow Example.

these tasks, is specially introduced to decouple training processes into client processes and server processes. However, just-in-time checkpointing's framework directly uses the device proxy of Singularity [31], which is specially designed for elastic training, not for fault tolerance. Therefore, many redundant features, such as transparent migration and elasticity, are all reserved in this device proxy. These redundant features lead to overheads and inefficiency, and may also introduce new sources of software failure. As evaluated in §4, Singularity's device proxy brings 7.1% daily overhead, which should not be underestimated considering clusters' huge scale.

Second, for overhead in recovery work, resource reinitialization is the most time-consuming part. Due to just-in-time checkpointing's design, healthy nodes' computation resources, including training states, model parameters, and optimizer states, can be fully retained when errors occur. Only error nodes need to copy these computation resources from DP replicas after restart. However, communication resources are not considered in recovery, which means the communication backends (e.g., NCCL) of all nodes will be aborted and then reinitialized, regardless of whether the node is affected by errors. Unfortunately, communication backend reinitialization consumes considerable time in recovery, which can even take up to 90% of the entire restart process [11] (Table 1). Meanwhile, today's mainstream communication backends, such as Gloo [21] and NCCL [24], do not support runtime modification, which results in a fixed communicator once created and also brings difficulties to the design of the upper-level fault-tolerant framework.

From the analysis above, we can see that while just-in-time checkpointing introduces new directions for fault tolerance in large-scale LLM training, its practical efficiency remains bottlenecked by unoptimized legacy components and naive communication reinitialization strategies. The inherent contradiction between its lightweight design philosophy and inherited infrastructure bloat from Singularity creates a paradoxical scenario: the framework aims to reduce overhead yet inadvertently reintroduces latencies through non-specialized device proxies, and the overlooked temporal dominance of communication backend reinitialization during recovery undermines its key advantage of swift node resurrection. These insights motivate a dual-axis redesign: (1) disentangling essential fault-tolerance mechanisms from elastic training features in device proxies to achieve true transparency and efficiency, and (2) pioneering partial or incremental communication reinitialization techniques compatible with CCLs and deep learning frameworks.

3 Mnemosyne Design

To address the problems above, as shown in figure 2, we propose Mnemosyne, a lightweight and fast error recovery framework for

LLM training. Mnemosyne introduces the device proxy specially designed for error recovery and the flexible CCL capable of dynamically adjusting links in the runtime (Figure 2(a)), fully unleashing the efficiency potential of just-in-time checkpointing. When errors occur, it can provide efficient error recovery via state migration and link adjustment (Figure 2(b)), which is fully transparent to upper frameworks.

3.1 Error Recovery Workflow

Similar to just-in-time checkpointing [11], Mnemosyne ensures error recovery for LLM training through three sequential phases: steady-state operation, error interception, and recovery execution. These phases collectively enable lightweight state tracking, user-transparent fault detection, and local restoration while maintaining minimal overhead during normal execution. Here we take a simple training task with a DP group of 4 nodes (shown in figure 3) as an example to show the workflow of Mnemosyne.

Steady-State Operation. During normal training iterations, Mnemosyne maintains a replay log to record all device API activities, e.g., CUDA kernel launches, NCCL communications (Figure 3(a)). The log stores API parameters, GPU object handles (e.g., streams and events), and buffer addresses. Log record is taken by *device proxy*, a carefully designed component able to decouple training jobs from devices and support job migration (§3.2). Due to our efficient design, this log brings almost zero overhead to normal training process. At each minibatch's boundary, the log is cleared to reduce memory consumption. This guarantees that logged inputs are complete enough to deterministically recover GPU states while not taking up too much space.

Error Interception. The device proxy continuously monitors API return codes and system-level signals to detect failures. GPU-related errors (e.g., invalid memory access) are immediately captured via API wrapper hooks. Upon detection, the layer halts API propagation to the framework, isolates the faulty GPU, and triggers the recovery pipeline.

Recovery Execution. Recovery execution are dynamically selected based on error types. For recoverable software or driver errors, we can directly recover in space (Figure 3(b)). While for irrecoverable errors (e.g., hardware faults), we can recover with migration (Figure 3(c)). In this case, the CRIU [5] tool snapshots CPU process states (e.g., memory, file descriptors), while model parameters and optimizer states are pulled from DP replicas. After migration, the system restores the GPU execution context using the replay log from the last completed minibatch. After these steps, training task's CPU state is restored to make errors imperceptible to the upper framework. This includes GPU-related handle remapping by our device proxy, and fast communication rebuild via dynamic link adjustment by our *flexible CCL* (§3.3).

3.2 Device Proxy

Different from existing device proxy [11, 31] that prioritizes transparent migration and elasticity through full virtualization mechanisms (e.g., fully GPU memory management, handle virtualization, barrier, time slicing, etc.), our device proxy focuses on fault tolerance with minimal runtime overhead, pursuing lightweight and efficiency. It introduces three core mechanisms: (1) GPU context decoupling, (2) CUDA-related call interception, and (3) index-based handle mapping.

GPU context decoupling. To achieve transparency, error and recovery details should be hidden to upper frameworks, which requires unnoticeable restart, migration, and GPU operations. Therefore, the GPU context of training jobs should be decoupled from the training process. In our design, the device proxy splits a single training job into two processes, a client and a server, since standalone processes naturally provide resource isolation. In this dual-process architecture, clients are responsible for the proceeding of training jobs, while servers handle all GPU operations committed by clients. Each worker's proxy client and server are one-to-one corresponding (Figure 2(a)), different from Singularity's design where the correspondence is one-to-n with elasticity taken into consideration. This difference brings lightweight to our device proxy's resource management, thus leading to higher efficiency of decoupling.

Although decoupling is convenient for error recovery, it brings extra communications, so efficient IPC channels for clients and servers are of great necessity. In Mnemosyne, IPC is carried out via a message queue based on shared memory, optimizing performance while ensuring functionality. Specially, for some APIs like `cudaMemcpy`, servers need to obtain massive information from clients. For this case, our solution implements a pipelined transferring mechanism. It first divides source data from client memory into fixed-size blocks and then transfers in order, so sending and receiving are overlapped and can be carried out at the same time, hugely cutting down time consumption.

CUDA-related call interception. Since GPUs are taken over by the clients, CUDA-related function calls should be redirected to servers for execution. Therefore, all CUDA-related calls should be intercepted by clients and then forwarded to servers. Our clients achieve such interception via `LD_PRELOAD` [20], a mechanism provided by Linux for runtime stubbing. After intercepting a call, its function identifier and parameters are serialized and then committed to the server via the IPC channel. The server executes the operation accordingly, and then give the execution result back to the client. Meanwhile, the separation of interception and execution inherently provides error isolation between CPU and GPU states. The server monitors operations' return values to detect GPU-side errors. When errors occur, they are contained within the server process using three-stage containment: (1) Error state capture through CUDA API return code validation; (2) Context quarantine via immediate server-side resource release; (3) Transparent error recovery as described in §3.1. This separation ensures that training frameworks never observe GPU errors directly since clients remain valid during server-side restarts or resource migrations.

Index-based handle mapping. During restart or migration, the handles of allocated resources in GPU side are altered due to replay. Therefore, raw handles should not be directly exposed to upper training frameworks since it may be changed during recovery. In our device proxy, an index-based handle-mapping array is maintained by the client, whose indexes are logical handles and elements are physical handles. Only necessary handles, such as GPU memory pointers (e.g., allocated memory by `cudaMalloc`) and GPU-related objects' handles (e.g., `cudaStream` and `ncclComm`), are recorded in our mapping to reduce overheads. During API call translations: (1) Client-side logical indexes are converted to server-side physical addresses, (2) Execution occurs on physical resources by the server, and (3) Returned physical addresses are re-wrapped as logical indexes to be provided for clients. Both record addition and lookup is of $O(1)$ time complexity since they are the simplest array appending and indexing operations with almost zero overhead. When replay is carried out, the array can be updated sequentially due to the consistency of original and replayed operations' orders.

3.3 Flexible CCL

Our flexible CCL is designed based on the incremental development of NCCL [24]. Through carefully designed APIs and mechanisms, i.e., metadata-driven node initialization and runtime communication modification, flexible CCL maintains backward compatibility with existing NCCL APIs, and introduces two novel features for today's CCL, immediate node replacement and dynamic scalability.

Compatibility-preserved efficient APIs. To preserve compatibility with NCCL, we design new NCCL APIs in an incremental way, introducing new features with no modifications to definitions and usages of original APIs. Newly added APIs on node adjustment are all executed by device proxies' server side, requiring no modifications to the training frameworks. For node replacement and addition, we break down the operations into 4 stages: (1) *Exporting metadata.* Flexible CCL needs built communicator's metadata for new node's initialization, so required metadata are exported from any healthy node via API `ncclCommExportInfo`. (2) *Initializing new node.* With exported communicator's info, the new node completes initialization via APIs like `ncclInitNewNode`. (3) *Updating communicator's metadata.* Already existed healthy nodes use APIs like `ncclReplaceNode` or `ncclAddNode` to update themselves' metadata with the ones from the new node. (4) *Build channels.* Needed channels are build according to modifications to the communicator via API `ncclCommSetupNewRank`. For node removal, we introduce one API `ncclRemoveNode` since initialization of new node is not involved in this case.

Metadata-driven node initialization. Today's CCLs mostly initialize communicators in a synchronized manner, where involve rounds of global information exchange and coordination (e.g., host-port pair collection, topology profiling). Such procedures are clearly unsuitable for dynamic localized initialization of flexible CCL. Different from conventional CCLs, flexible CCL initializes newly introduced node with the built communicator's metadata, which is based on our observation that initialized nodes inherently keep complete communicator's metadata within their `ncclComm` structures.

During the export of metadata, there are two different categories of exported metadata: raw collected data, which is completely stored in arrays as gathered (e.g., host-port pairs), and reduced data, of which only one copy of reduced result is stored after gathered (e.g., aggregated bandwidths). For raw collected data, we directly copy the existing node's memory structure from exported metadata. While for reduced data, we use partial re-reduction: the replacement node combines local computed data with the received reduced value, exploiting the associativity of reduction operators. This bypasses the need to reconstruct original per-node data while maintaining result consistency. This strategy is also applicable for the whole communicator's metadata update.

For topology construction, we observe that NCCL optimizes only intra-machine device connections due to scalability consideration, so we extend this strategy for node replacement. When substituting failed nodes, the new node directly takes up original node's position in the topology, and rebuilds links connected to the original node. Meanwhile, because our replacement granularity is node, the optimality of intra-node's topology is guaranteed by NCCL's calculation. This hybrid approach maintains native performance levels for intact communication paths. In the future, we are planning to design a more flexible topology construction strategy for more universal fault tolerance.

Runtime communicator modification. Flexible CCL supports runtime communicator modification. First, runtime communicator expansion requires careful memory management to accommodate rank additions. Existing NCCL implementations statically allocate memory buffers for peer addresses and topology data, creating runtime fragmentation risks during resizing. Our solution pre-allocates configurable buffer space during initial communicator creation. Reserved slots enable seamless insertion of new ranks without memory reallocation, while pointer stability gets ensured through contiguous memory layouts and offset-based addressing. Second, node addition and removal also bring modification to topology. Currently, the modification strategy for ring topology is designed. For an added node, flexible CCL directly connect it to two originally adjacent nodes. For a removed node, flexible CCL directly connect its previous node to its next node. In future work, we will design strategies for more topologies, e.g., double binary tree.

4 Implementation and Evaluation

We implement a prototype of Mnemosyne to verify our design's feasibility and efficiency. The device proxy can support CPU and GPU context isolation for distributed training tasks, containing over 17,000 lines of C++ source code. The flexible CCL is implemented based on NCCL with over 1,700 lines of C++ source code modification, which supports addition or removal of a single rank from built communicators. Relevant implementations are now open source [2]. Our experiments are carried out on a node with 3 NVIDIA Quadro RTX A6000s connected via PCIe.

Device proxy. To evaluate the efficiency of our device proxy, we use small-scale Transformer [33] models with various architectures, including encoder-only model (e.g., BERT [6]), decoder-only model (e.g., GPT [26]), and encoder-decoder model (e.g., T5 [28]). Figure 4 shows that the average overhead of our device proxy during steady-state operation is 3.6%. Compared with Singularity, the performance overhead is reduced by 49.1% on average, 58.8% at most. Notably, since our current experiment is conducted on relatively small LLM models, the API call frequency is relatively high, and the overhead

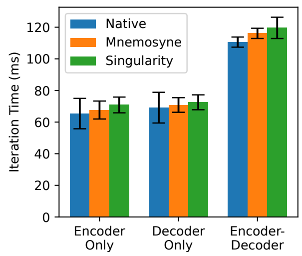


Figure 4: Daily overhead of our device proxy.

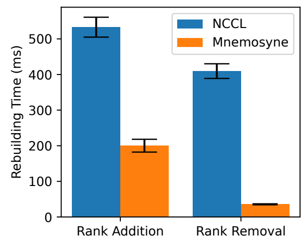


Figure 5: Acceleration effect of our flexible CCL.

of the device proxy becomes higher accordingly. In practical deployments of large-scale LLM models, the kernel computation time of these models is much longer and the API call becomes less frequent, so the relative overhead of the device proxy will be significantly reduced.

Flexible CCL. To validate the availability and efficiency of our flexible CCL, we select rank as the granularity of operations instead of node due to the limitations of our experimental environment, and prioritize addition than replacement since addition not only includes initialization and update operations of replacement, but also involves more complex metadata adjustments. Rank addition adds 1 rank to a communicator of 2 ranks, and rank removal removes 1 rank from a communicator of 3 ranks. Figure 5 shows that compared to NCCL, the flexible CCL has significant improvement in runtime adjustment, taking an average of 8.7% of global reinitialization time for removal and 37.6% for addition.

5 Discussion

Limitation on FSDP. The Fully Sharded Data Parallel (FSDP) approach, such as DeepSpeed ZeRO-3 optimizer [29], reduces GPU memory consumption through distributed parameter sharding across data parallel groups. In training scenarios employing FSDP methodology, the absence of redundant model parameters precludes the application of Just-In-Time checkpointing techniques, thereby rendering it incompatible with Mnemosyne.

Training anomaly detection. Mnemosyne is a framework specially designed for error recovery of LLM training, and does not pay attention to training anomaly detection and location. While our framework can detect GPU errors via execution result of CUDA operations, additional mechanisms are required to locate the root cause of some subtle anomalies, e.g., stragglers and framework hang [15]. Such works are orthogonal to our work, which can be combined together to achieve more efficient fault tolerance.

Network errors. Mnemosyne is designed for GPU errors, and network errors are overlooked to some extent. On one hand, most network switches and links have redundancy, which intrinsically supports fault tolerance. On the other hand, our strategy is also applicable for network errors, e.g., failures of network links connected directly to GPU nodes or ToR switches. For errors of network links, we can migrate jobs at the granularity of nodes, while for errors of Tor switches, we can migrate jobs at the granularity of racks.

6 Related Work

LLM training fault tolerance. Besides periodic checkpoint [7, 15, 22, 23, 35] discussed in the main text, another line of LLM training fault tolerance leverages the characteristics of pipeline parallelism in LLM training. Bamboo [32] redundantly computes two adjacent sub-stages on each GPU in the pipeline parallel training mode. Oobleck [13] employs heterogeneous pipeline templates and instantiates multiple logically equivalent pipeline replicas in large DNN models. ReCycle [9] utilizes pipeline bubbles to compute the tasks of the failed rank to reduce the resource overhead. However, these works incur redundant computation, decreasing the training throughput and risking running out of GPU memory. More importantly, when the training job needs to be restarted after a failure, they all require the global reinitialization of NCCL and incur significant startup overhead.

Collective communication libraries. In recent years, significant advancements have been made in optimizing collective communication, particularly within the context of LLM scenarios. Blink [34] leverages the heterogeneous communication channels of GPU clusters to optimize the performance of data transmission. OmniReduce [8] takes advantage of the sparsity of models to enhance bandwidth utilization. CoCoNet [14] jointly optimizes communication and computation GPU kernels, improving the performance of distributed workloads. SCCL [3], TACCL [30], TCCL [17], and TECCL [18] have made optimizations based on the topology in terms of programmability and communication performance. Mnemosyne is orthogonal to these aforementioned studies. By integrating these techniques, Mnemosyne is expected to achieve better performance while maintaining superior fault tolerance capabilities.

7 Conclusion and Future Work

In this paper, we identify the limitations of current checkpointing mechanisms, and design Mnemosyne, a lightweight and fast error recovery framework for LLM training. Two key components, device proxy and flexible CCL, are tailored for Mnemosyne to achieve lower daily and recovery overhead. Our prototype's implementation and evaluation demonstrate that Mnemosyne can bring more efficiency than today's frameworks in various aspects.

Nevertheless, our current design, prototype and evaluations are still very preliminary, which leaves a lot of future works to continue. In our ongoing explorations, we plan to further optimize the steady-state overhead, introduce a pipelined migration mechanism to accelerate the recovery process, extend flexible CCL's strategies for communicator modification to more topologies, design a mechanism for migration destination selection, refine the implementation of Mnemosyne, and evaluate our design in real large-scale LLM training scenarios.

Acknowledgments

We thank anonymous APNet reviewers for their valuable comments. This work is supported in part by the National Natural Science Foundation of China (No. 62402025), the Beijing Science and Technology Plan Project, the Fundamental Research Funds for the Central Universities and gifts from Huawei-BUAA Joint Lab. Menghao Zhang is the corresponding author.

References

[1] Wei An, Xiao Bi, Guanting Chen, Shanhuang Chen, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Wenjun Gao, Kang Guan, et al. 2024. Fire-Flyer AI-HPC: A Cost-Effective Software-Hardware Co-Design for Deep Learning. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–23.

[2] Anonymous authors. 2025. Mnemosyne. https://anonymous.4open.science/r/Mnemosyne

[3] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. 2021. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 62–75.

[4] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research* 24, 240 (2023), 1–113.

[5] CRIU. 2024. Checkpoint/Restore in Userspace. https://criu.org/Main_Page

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*. 4171–4186.

[7] 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)*. 929–943.

[8] Jiawei Fei, Chen-Yu Ho, Atal N Sahu, Marco Canini, and Amedeo Sapio. 2021. Efficient sparse collective communication and its application to accelerate distributed deep learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 676–691.

[9] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. 2024. ReCycle: Resilient Training of Large DNNs using Pipeline Adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 211–228.

[10] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[11] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. 2024. Just-in-time checkpointing: Low cost error recovery from deep learning training failures. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1110–1125.

[12] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. 2024. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 709–729.

[13] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. 2023. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 382–395.

[14] Abhinav Jangda, Jun Huang, Guodong Liu, Amir Hossein Nodehi Sabet, Saeed Maleki, Youshan Miao, Madanlal Musuvathi, Todd Mytkowicz, and Olli Saarikivi. 2022. Breaking the computation and communication abstraction barrier in distributed machine learning workloads. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 402–416.

[15] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. 2024. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 745–760.

[16] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[17] Baojia Li, Xiaoliang Wang, Jingzhu Wang, Yifan Liu, Yuanyuan Gong, Hao Lu, Weizhen Dang, Weifeng Zhang, Xiaojie Huang, Mingzhuo Chen, et al. 2024. TCCL: Co-optimizing Collective Communication and Traffic Routing for GPU-centric Clusters. In *Proceedings of the 2024 SIGCOMM Workshop on Networks for AI Computing*. 48–53.

[18] Xuting Liu, Behnaz Arzani, Siva Kesava Reddy Kakarla, Liangyu Zhao, Vincent Liu, Miguel Castro, Srikanth Kandula, and Luke Marshall. 2024. Rethinking machine learning collective communication as a multi-commodity flow problem. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 16–37.

[19] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, et al. 2021. Understanding and improving failure tolerant training for deep learning recommendation with partial recovery. *Proceedings of Machine Learning and Systems* 3 (2021), 637–651.

[20] Linux manual page. 2024. ld.so. https://man7.org/linux/man-pages/man8/ld.so.8.html

[21] Meta Platforms, Inc. 2023. Gloo: Collective Communications Library. https://github.com/facebookincubator/gloo

[22] 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)*. 203–216.

[23] 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)*. IEEE, 172–181.

[24] NVIDIA Corporation. 2023. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl

[25] OpenAI. 2024. Learning to reason with LLMs. https://openai.com/index/learning-to-reason-with-llms/.

[26] Alec Radford. 2018. Improving language understanding with unsupervised learning. *OpenAI Res* (2018).

[27] 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.

[28] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.

[29] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 20, 16 pages.

[30] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. 2023. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 593–612.

[31] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, et al. 2022. Singularity: Planet-scale, preemptive and elastic scheduling of AI workloads. *arXiv preprint arXiv:2202.07848* (2022).

[32] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2023. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 497–513.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[34] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. 2020. Blink: Fast and generic collectives for distributed ml. *Proceedings of Machine Learning and Systems* 2 (2020), 172–186.

[35] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS 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*. 364–381.

[36] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[37] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* 1, 2 (2023).