LATTICE: Efficient In-Memory DNN Model Versioning

Manoj P. Saha
Florida International University
Miami, FL, USA
msaha002@fiu.edu

Ashikee Ghosh
Florida International University
Miami, FL, USA
aghos008@fiu.edu

Raju Rangaswami
Florida International University
Miami, FL, USA
raju@cs.fiu.edu

Yanzhao Wu
Florida International University
Miami, FL, USA
yawu@fiu.edu

Janki Bhimani
Florida International University
Maimi, FL, USA
jbhimani@fiu.edu

Abstract

DNN model versions are used for various tasks such as fine-tuning for downstream tasks, explainability, and debugging. Numerous checkpointing solutions exist that can be adapted to persist intermediate versions of a model, as it is being trained, at different storage locations. Additionally, version management tools allow us to log, visualize, compare, and query metadata related to ML, tracking changes made to previously built models. However, the version creation process of existing methods incurs high runtime and storage overheads. In this paper, we introduce **LATTICE**, a low-latency, direct persistence-based DNN versioning library for Non-Volatile Memory (NVM) expansion devices. LATTICE minimizes stalls during model versioning and reduces end-to-end versioning time by reorganizing the version creation workflow, streamlining memory allocation and deallocation for efficient snapshot creation, and leveraging multi-threaded parallelism. We also develop a user-friendly versioning API that transparently implements direct persistence. Our comprehensive evaluation with diverse DNN models shows that LATTICE can reduce persistence time by as much as 99.99%, decrease end-to-end versioning time by up to 72%, reduce versioning stalls by up to 35%, and increase versioning frequency by 0.2×-3.84× compared to state-of-the-art solutions. LATTICE also reduces space utilization for different workloads. The space savings are from 23.8% to 43.2% for workloads where model layers are progressively frozen and from 84.8% to 98.9% for fine-tuning workloads where only the last layers are tuned.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SYSTOR '25, Virtual, Israel
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2119-9/25/09
https://doi.org/10.1145/3757347.3759139

CCS Concepts

• **Information systems** → **Storage class memory**; • **Computing methodologies** → **Machine learning**.

Keywords

persistent memory, non-volatile memory, deep neural network, checkpointing

ACM Reference Format:
Manoj P. Saha, Ashikee Ghosh, Raju Rangaswami, Yanzhao Wu, and Janki Bhimani. 2025. LATTICE: Efficient In-Memory DNN Model Versioning. In *Proceedings of ACM International Systems and Storage Conference (SYSTOR '25).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3757347.3759139

1 Introduction

Deep Neural Networks (DNNs) have become indispensable tools in a wide range of applications, encompassing image classification [15], language translation [36], and speech recognition [43]. While these models have significantly improved performance in various domains, training complex DNNs on extensive datasets often demands several days, weeks or months of computing resources [17, 25]. The trained model, along with its intermediate versions generated during training, serves a multitude of purposes. These include failure recovery [29], model analysis [7], transfer learning [48], neural architecture search [44], snapshot ensembles [16, 41], fine-tuning of models for various downstream tasks [20, 46], online training [47] and progressive training [4].

Despite the central role of DNN model versioning in deep learning (DL), designing a versioning solution that is efficient, correct, and reliable introduces its own challenges. First, it is important to ensure the correctness and consistency of persisted model versions for multiple reasons. For example, if a training job recovers using an inconsistent model version it can lead to noticeable drops in accuracy upon restart,

leading to higher training time. In some cases, the training job may fail to recover as well, resulting in a restart with an older consistent version of the model that can lead to increased training time. Second, to reduce the amount of wasted work due to software or hardware induced DNN job failures [11, 26], state-of-the-art solutions persists the model state frequently [6, 13, 28, 31]. This leads to frequent stalls and adds to the overall training time. Third, if all or a substantial part of these model versions are required to be preserved, as is the case for model analysis or Neural Architecture Search (NAS), it can lead to high space overhead. Hence, checkpointing mechanisms that persist as little as possible to reduce space usage [42] cannot be used for versioning. Fourth, due to the high file I/O overhead associated with traditional checkpointing solutions [13, 28], using them for versioning would increase training time. Fifth, modifying existing PM-based checkpointing solutions [6, 24, 31] for versioning purposes is not straightforward due to challenges such as memory allocation/deallocation, buffer management, and the need to know the number of versions in advance. Naïvely enabling support for multiple model versions in GPM [31] or LightCheck [6] will lead to increased runtime and storage overheads. Lastly, state-of-the-art version management tools, such as ModelDB [39], DVC[3], MLflow[45], and Git-LFS [14], enable efficient tracking of model evolution by logging, visualizing, comparing, and querying metadata, facilitating the management of changes made to previously trained models. However, these tools rely on file I/O for persistence and are not designed for in-memory versioning.

In light of these observations, we assert the fundamental need to optimize the overall versioning process for DNN models using direct persistence. In this paper, we introduce, **LATTICE**, a **L**ow l**AT**ency direc**T** pers**I**sten**C**e v**E**rsioning library for DNN workflows. Our major contributions are as follows:

- We design a novel workflow to minimize DNN model versioning time and introduce the concept of *eager versioning*.
- We create a user-friendly versioning API that transparently realizes direct persistence.
- We optimize the DNN snapshot creation process by reducing memory allocation and de-allocation overheads.
- We harness the internal parallelism of non-volatile memory expansion devices with multi-threaded version creation.
- We leverage operating system building blocks to reduce space utilization during version creation.
- We evaluate LATTICE in comparison with state-of-the-art methods.

We implement LATTICE as a pluggable module for DNN frameworks. Our evaluation across various models, datasets, and storage devices shows that LATTICE outperforms state-of-the-art solutions, reducing model version persistence time by up to 99%, minimizing end-to-end versioning time by 72%, and cutting versioning stalls by 35%. Additionally, LATTICE enables 20%–384% more frequent checkpoints for large models compared to file I/O-based versioning methods. LATTICE reduces space utilization from 23.8% to 98.9% for workloads where only some of the model parameters or layers are updated in subsequent iterations.

2 Extended Motivation

Beyond Failure Recovery. DNN model versioning captures the entire state of the model being trained, at various points in time as needed. These versions are all available in persistent storage and can be restored at any time for further use. DNN failure recovery solutions use the latest persisted version of the model to reduce lost work due to failures. Recent checkpointing solutions such as CheckFreq [28] delete any older model versions, as soon as a new version is persisted, to reduce space utilization. In-memory checkpointing solutions use double-buffering and overwrite the oldest among two most recent versions to save space. However, it is not straightforward to re-purpose checkpointing solutions to general purpose model versioning solution. For example, GPM [31], LightCheck [6] and Portus [24] cannot be used for offline model analysis jobs that requires multiple versions of the model (see Table 1). However, use cases such as defense against adversarial attacks [12], online training [13], model fine-tuning [20, 46], transfer learning [48] that need to maintain multiple model versions are prevalent in DNN model lifecycle [45]. While file I/O-based model versioning solutions [28] can be easily retrofitted to support multiple model versions, such versioning schemes are not optimal and not inherently space-efficient. This is due to comparatively larger persist time of file I/O on persistent storage and lack of data sharing between model versions. The above observations suggest that there is a fundamental need to realize an efficient DNN model versioning solution to serve different downstream tasks.

NVM Expansion Devices in DNN Training. Memory expanders, such as Intel Optane and CXL-attached NVM devices, are particularly well-suited for ML versioning and checkpointing due to their low-latency access and persistence properties. These devices enable faster data access compared to traditional storage solutions like SSDs and HDDs. CXL-attached NVM devices can also help in emerging use cases where model versions are deployed to defend against adversarial attacks [12]. Thus, we believe that designing an

Table 1: Comparison of existing DNN checkpointing solutions [I/F=Interface; F=File; M=Memory; GP=General Purpose; FR=Failure Recovery; RM=Recommendation Model; ** to be open sourced at https://github.com/Damrl-lab/LATTICE].

Solution	I/F	Multi. Version	Space Optim.	Use Case	Open Source
CheckFreq[28]	F	X	X	FR	✓
Check-N-Run[13]	F	✓	✓	RM	X
GPM[31]	M	X	X	X	✓
LightCheck[6]	M	X	X	X	✓
Portus[24]	M	X	X	FR	X
LibPM[27]	M	X	X	FR	✓
LATTICE	M	✓	✓	GP	✓**

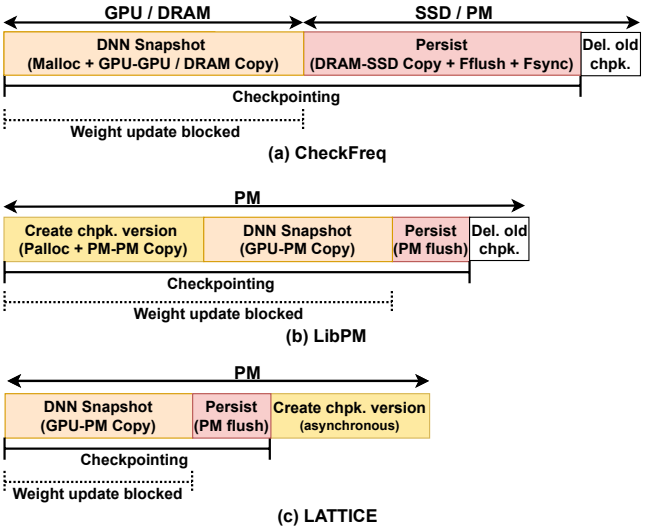


Figure 1: Checkpointing workflow in different libraries.

efficient in-memory DNN model versioning solution that seamlessly works with NVM expanders has great potential.

Limitations of Existing Checkpointing Solutions for Model Versioning. Realizing an efficient DNN model versioning solution that optimizes both runtime and space overheads is non-trivial. While it is easier to retrofit existing file I/O-based checkpointing solutions such as CheckFreq [28] to support DNN model versioning, the file I/O stack overheads make it sub-optimal when used with NVM expansion devices [24]. Re-purposing Check-N-Run's [13] consecutive incremental differential checkpointing technique to realize space-efficient model versioning is challenging. Restoring from consecutive incremental differential checkpoints mandates copying all previous checkpoints from storage to CPU DRAM, requiring significant bandwidth. On the other hand, existing DNN checkpointing solutions designed for NVM expansion devices, such as GPM [31], LightCheck [6], and Portus [24], are tailored specifically for failure recovery. These solutions are designed to maintain only the latest consistent model state and are based on double buffering/mapping and a zero-copy mechanism. That is, non-volatile memory regions are pre-allocated in the NVM exapnsion device to hold two versions of the model at any point in time, a working copy and a consistent copy, both equal in size. This eliminates memory allocation overheads during a checkpoint operation and also eliminates data copy between the working copy and consistent copy. When a new persist operation is triggered, the direct persistence approach persists the memory pages containing the model parameters in the working copy at near-memory latency. The working copy is then marked as consistent copy and vice versa. Extending these solutions to support DNN model versioning beyond the latest consistent model state is non-trivial. We would need prior knowledge of the number of versions required by the ML application to pre-allocate the buffers. Furthermore, a solution unaware of data sharing opportunities across model versions that simply stores entire model versions independently would be very space-inefficient. For example, for a model fine-tuning job that only tunes the last layer of the model, existing solutions would be writing out and storing two identical copies of all the other layers, wasting valuable memory capacity and bandwidth. Realizing space-efficient versioning solutions by extending the above mentioned solutions would need non-trivial changes in their internal data structures and workflow.

Versioning Library. An alternative is to build a DNN model versioning library on top of an existing library designed for byte-addressable NVM expansion device, which allocates only the required amount of space during each persist operation. Existing libraries designed for byte-addressable NVM expansion devices such as Mnemosyne [40], NV-Heaps [8], LibPM [27] and Pmem.io [32] target making CPU-accessible DRAM data structures persistent consistently [2]. They are not designed to optimally persist data that is generated in off-CPU accelerator devices such as GPUs. For example, PMEM.io uses locks and undo log to ensure consistency of data (i.e., model being trained in this context). On the other hand, LibPM [27] follows a copy-on-write mechanism to create a consistent copy of the model, once a commit operation is triggered (Figure 1b). Both of these methods incur additional memory allocations and NVM device-internal data copy in the critical path, which increases runtime overhead of creating a consistent model version. More importantly, none of these schemes support versioning. Existing solutions can only store a single consistent version of the model, at any point in time.

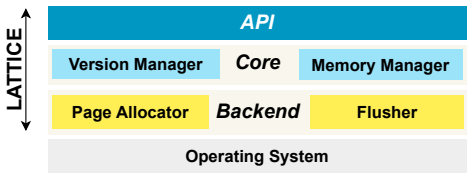


Figure 2: LATTICE Architecture.

API. When developing a DNN model versioning library based on byte-addressable NVM expansion devices, it is important to provide ML application developers a simple and streamlined application programming interface (API) that can transparently create, manage and restore model versions. In addition, the space-efficient versioning, where only the updated portions of the model state gets persisted, requires us to redesign the version restore workflow to transparently splice different model versions to re-create a full model. It is also important to make the following operations transparent to the ML applications - metadata management of different versions, creation of checkpoints with partial or full model state updates, and restore of checkpoints from full or spliced model state version(s). LATTICE offers easy-to-use APIs to create and restore versioned checkpoints using a *persistent container abstraction.*

3 Architecture and Design Principles

LATTICE is a DNN model versioning library designed for byte-addressable non-volatile memory (NVM) expansion device. It exploits *direct persistence* [21] to allow ML applications maintain persistent model versions and restore DNN models at the speed of memory. LATTICE's primary goal is to optimize in-memory DNN model versioning process, transparently manage model versions in space-efficient persistent containers (i.e., an in-meomory data storage unit that ensures consistency and integrity of arbitrary data types) and enable ML applications to efficiently restore versions. This is achieved by reducing runtime overheads of persisting model versions, exploiting data sharing opportunities between versions, and transparently managing pointer consistency while restoring model versions. To realize this, LATTICE reorganizes different stages of the in-memory versioning workflow, implements a lightweight mechanism to identify data sharing opportunities, and uses a range of data structures to manage data consistently inside persistent containers. By tailoring DNN model versioning stages to the specific needs of DL applications, minimizing tasks performed in the critical path and carefully organizing data and metadata inside the persistent container, LATTICE ensures that the entire process is streamlined and efficient.

3.1 LATTICE Architecture

LATTICE's internal architecture is divided into two sets of components - *Core* and *Backend* (Figure 2). The Core provides transparent memory allocations and management, and persistent version creation support to DNN applications using two interconnected subcomponents - *Memory Manager*, and *Version Manager*. The Memory Manager is responsible for automatically discovering the entire memory footprint of the DNN model instance created in DRAM by the application, and transparently moving these memory allocations from DRAM to LATTICE container. This data movement from volatile DRAM to the byte-addressable NVM-resident LATTICE container is done exactly once during initialization so that model synchronization can then happen directly between GPU memory and NVM for all subsequent versioning operations. The *Version Manager* manages the versioning process and the various persisted versions of the DNN model. The Backend consists of Page Allocator, which maps and manages data within non-volatile memory pages, and Flusher, which persists data in non-volatile media.

3.2 LATTICE Design Principles

LATTICE design is driven by three overarching goals —

- *Reduce training stalls by eliminating operations from the critical path.* State-of-the-art byte-addressable NVM libraries like LibPM, when used for DL checkpointing, use in-place updates and start by creating a read-only version (i.e., a consistent copy) of the previous checkpoint as soon as a new checkpoint operation is requested. This read-only version remains until the new checkpoint is persisted, which involves copying data from the GPU to the NVM, and NVM flush (platform dependent) and fence operations. We have observed that this checkpoint version creation process happens in the critical path repeatedly during training, leading to *training stalls.*. To address this issue, LATTICE introduces *eager versioning*, a proactive approach that creates versions based on a *copy-before-write* model. *Eager versioning* enables LATTICE to significantly redesign the sequence of operations required for model versioning. Unlike the conventional approach used in LibPM (Figure 1b), where versioning takes place at the beginning of the checkpointing process, LATTICE shifts this operation to occur after the checkpointing is finished (Figure 1c). This rearrangement enables LATTICE to reduce training stalls by unblocking foreground training operations early.
- *Increase space efficiency by exploiting data sharing opportunities.* LATTICE employs in-place updates with write protection to maintain the model's consistency

Table 2: LATTICE API.

Category	API	Description
Container Management	init(...)	Create container
	delete(...)	Delete container
Annotation	setroot(...)	Set object as root of container
	getroot(...)	Get root object address
Version Creation - Restore	commit(...)	Persist model and create read-only version
	restore(...)	Restore specified version

throughout the checkpoint process. The write protection mechanism keeps track of non-volatile memory pages updated in the working copy since the last version. This lightweight mechanism of tracking changes between model versions is used to copy only the updated portions of the model to create a new version, while the unchanged portions of model are reused from the previous version. This strategy can *dramatically reduce* the amount of non-volatile memory pages used to create a DNN model version for certain workloads (e.g., model fine-tuning).

- *Exploits parallelism to reduce versioning latency of operations when possible.* LATTICE performs data copy and flush operations using multiple parallel threads inside its container during the version creation operation.

4 API and Design

DNN models are stored directly within persistent LATTICE *containers*. Containers represent a self-describing set of data that can be auto-discovered for persistence starting from a user-specified *root* structure. The container abstraction and accompanying automatic container data discovery mechanisms are borrowed from LibPM [27]. A single DNN model is stored within a single LATTICE container in this work. More than 1,600 lines of code has been added on top of LibPM to implement LATTICE.

4.1 LATTICE API

LATTICE's API, summarized in Table 2, replaces the file I/O interface for model versioning with a memory-oriented interface. The API calls can be grouped into three categories: container management, annotation, and version creation-restore. Container management interfaces are used to create and delete containers. To enable DNN model versioning, a training job would create a container using the *init()* call, which takes a number specifying that container as input. A training job can use multiple containers throughout the training process, if needed. A container can be deleted using the *delete()* API call.

At the beginning of the first version creation operation, we create a DNN model object that points to all the layers and parameters (i.e., tensors holding the model parameters in GPU) that'll be persisted in the LATTICE container. The DNN model object inside the LATTICE container mimics the DNN model object in GPU but purposefully omits pointers to tensors that won't be persisted (e.g., tensors holding intermediate states). This way, LATTICE's automatic memory discovery feature does not copy unintended tensors from the GPU during a version creation operation. The DNN model object is then set as the *root* object of the container using *setroot()* which creates a persistent (restorable) copy of the *root* object within the container. The *setroot()* operation to set the DNN model object as the container *root* object is needed only during the first version inside a container. During a version creation, LATTICE automatically discovers the data (i.e., tensors holding model parameters) by following the pointers in the container *root* object (i.e., the DNN model object) [27] and transparently performs the necessary non-volatile memory allocations within the container. The developer does not need to track modifications to model state in their code at all and this helps reduce model versioning latency and development complexity.

To create a persistent version of a model, the training application's *DNN snapshot* operation first copies model parameters and other metadata from the GPU to the LATTICE container. A *commit()* API call, which takes the container number as input, persists the model state as reflected in the *model directory* inside the LATTICE container and transparently creates an read-only version of the model under the *version directory*. LATTICE transparently optimizes the space utilization of the read-only model version by exploiting data sharing opportunities with older versions. During the restore operation, the training application retrieves the *root* object of the container using *getroot()*, and then uses *restore()* to load a specific DNN model version. Pointer consistency is maintained transparently within the LATTICE container while restoring any of the persisted model versions. If a model version number is not provided during a *restore()* API call, then LATTICE restores the most recent model version.

4.2 LATTICE Design

4.2.1 Versioning Process. A typical DNN training process is initiated on the CPU where a training thread sends the data and compute instructions to GPU. Figure 3 provides a high-level overview of how the DNN model versioning operations are handled inside a LATTICE container. A LATTICE container is initialized at the beginning of training using the *init()* API ① (Figure 3a). During initialization, the LATTICE container creates a *Model Directory*, a *Version Directory* and a *Free Page Directory*. The *Model Directory* contains a map of memory allocations of the latest version. The *Version Directory* maintains the list of read-only model versions stored

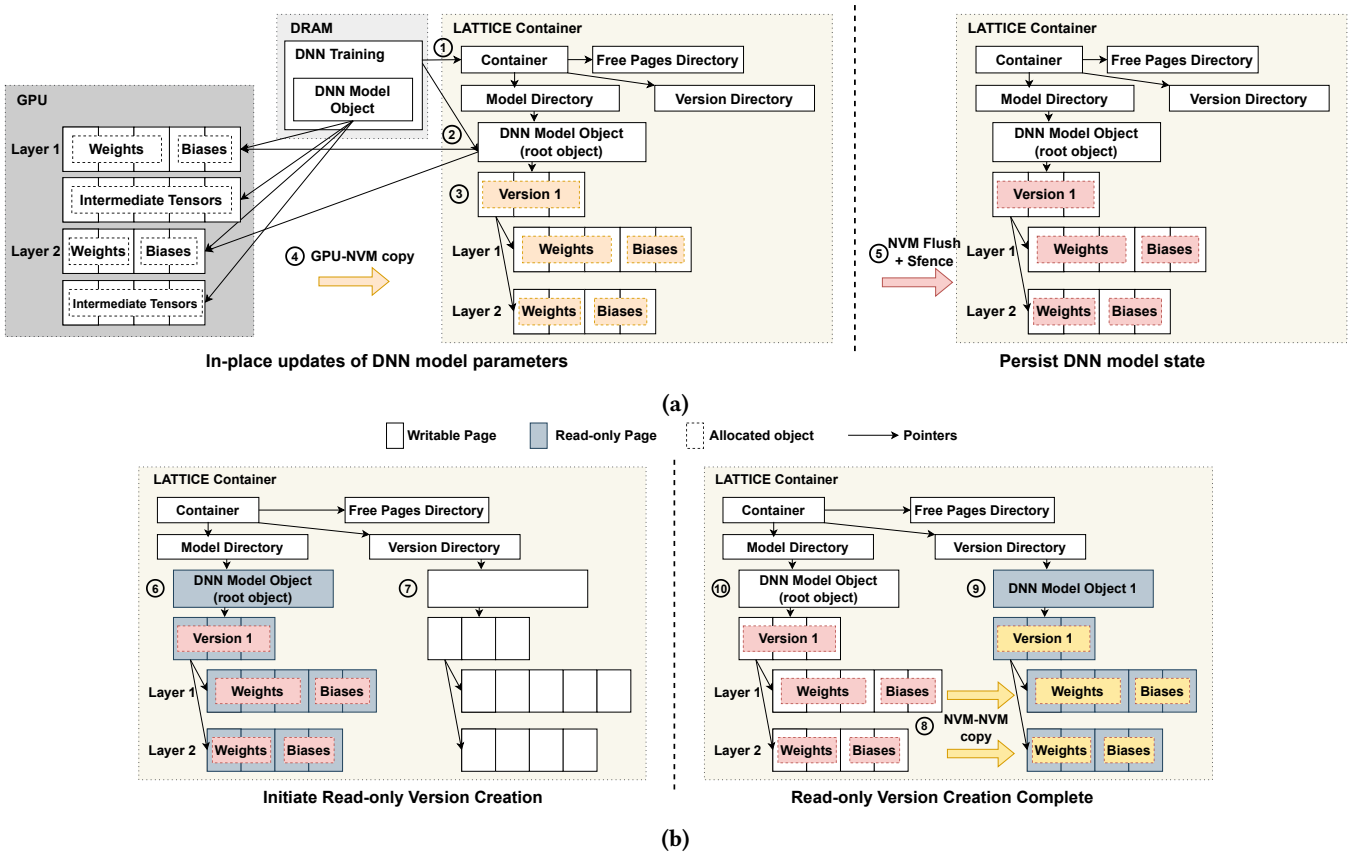


Figure 3: (a) LATTICE's in-place update feature reduces the time to copy model parameters from GPU and enables early persistence through simple PM flush cache operation, without incurring additional delay for DRAM-storage media data copy. (b) LATTICE reduces stall time in the future by preemptively creating DNN model versions.

inside a LATTICE container. Lastly, the *Free Page Directory* maintains the list of free pages available inside the container. Next, a *DNN model object*, which contains the pointers to the GPU memory pages to be persisted, is created and set as the *root* object ②. The *DNN model object* (i.e., the *root* object) reflects the model's architecture and contains pointers to the tensors holding the model parameters to be persisted. When the first version creation is issued, LATTICE's *Page Allocator* transparently allocates the non-volatile memory pages needed to copy the model parameters from GPU to the LATTICE container stored in the byte-addressable NVM expansion device ③. Next, the memory pages containing the model parameters are directly copied to the LATTICE container ④. Next, a NVM flush cache operation along with SFENCE, performed by the *flusher*, ensures that all updates to the LATTICE container are made persistent ⑤. This operation can be streamlined in CXL-attached NVM expansion devices (i.e., CXL Type 3 devices) with Global Persistent Flush (GPF) support [9], equivalent to eADR in Intel Optane [10]. Since systems with GPF support can flush cache lines to persistent NVM region in case of a power failure, regular cache flush operations can be avoided. Instead, data can be persisted using SFENCE operation only, which maintains correct write ordering required to ensure model consistency. If Direct Data I/O (DDIO) feature, in case of Intel CPU, or Direct Connect Architecture (DCA) feature, in case of AMD CPU, is turned off, the __threadfence_system() instrcution is needed to be issued from the GPU to maintain write ordering [31]. This operation persists the model inside the LATTICE container and completes the version creation.

The above method streamlines versioning in two ways. First, LATTICE avoids repeated memory allocations and deallocations during the DNN snapshot phase (i.e., copying model state from GPU to NVM) by providing in-place updates. The same NVM memory pages allocated under the *Model Directory* inside the LATTICE container are used to copy DNN model parameters from GPU to NVM during every subsequent version creation process. LATTICE's in-place updates reduce DNN snapshot time, despite GPU-NVM data copy incurring higher latency than GPU-DRAM data copy.

Intuitively, LATTICE's in-place updates incur similar latency of double buffering-based snapshots on NVM expansion devices. Avoiding DDIO or DCA feature during snapshot can further optimize snapshot time [6, 31]. Second, LATTICE leverages *direct persistence* and avoids DRAM-storage media data copy and high-latency *fsync()* operations for file I/O-based model versioning. Instead, a model version can be persisted by simply flushing modified cache lines to ensure all updated memory pages are successfully copied from the GPU to the LATTICE container.

4.2.2 Workflow Reorganization. Complementarily, LATTICE reorganizes the order of version-related operations to reduce training stall during in-memory versioning. To understand how such a reordering can reduce stall during model versioning, let us examine the conventional checkpoint creation workflow using single buffer in-place updates first. As shown in Figure 1b, updating model parameters under LibPM's *DNN Model object* for all subsequent versions is preceded by a version creation operation. This is necessary to ensure that a consistent version of the previous checkpoint is available if the system crashes during the current checkpoint. Once the read-only version creation completes, only then can the GPU-PM copy and persist operations proceed, thereby contributing to checkpoint stall time and overall checkpointing time.

To reduce the time taken for checkpointing, LATTICE introduces a significant improvement by reorganizing the version creation workflow, which includes different stages such as DNN snapshot, persistence, and read-only space-optimized version creation, as depicted in Figure 1c. Instead of creating a read-only copy of the previous checkpointed model state at the beginning of each version creation operation, LATTICE proactively generates a read-only version of model state immediately after it is persisted. We refer to this approach as *eager versioning*. Eager versioning creates persistent copies of checkpointed model version in parallel with model training prior to the next model version creation operation. Eager versioning's *copy-before-write* persistence model effectively enables model state to be persisted in-place immediately without having to block for a read-only version to be created first. In effect, *eager versioning* reduces training stall time.

The read-only space-optimized version creation operation is transparently started inside the LATTICE container following the persist operation in ⑤. To create a read-only version, the non-volatile memory pages storing the *DNN model object* are set as read-only ⑥. Next, LATTICE identifies and allocates new non-volatile memory pages to copy the updated portion of the *DNN model object* under *Version Directory* ⑦. This is followed by copy operations that copy the read-only pages to the newly allocated pages ⑧. Once the persistent copy operation completes, the non-volatile memory pages storing the *DNN model object* copy are made read-only ⑨ and the memory pages under the *Model Directory* are made writable again ⑩ allowing in-place model parameter updates. If a new version creation operation is triggered using the commit() API before the completion of operation ⑩, then the next GPU-NVM copy ④ operation will get delayed until ⑩ completes. Such scenarios can occur when applications persist relatively large model versions at the end of each iteration, and the version creation time far exceeds the iteration time.

4.2.3 Optimized Space Utilization. LATTICE uses per-page dirty bits to identify modified non-volatile memory pages to include in an incremental model version. If a non-volatile memory page does not get updated, LATTICE won't allocate and copy that page in the ready-only model version. Instead, the metadata structure containing the memory locations of the read-only model version would point to the corresponding memory location in the previous model version. If 50% of the memory pages in a particular checkpoint do not get updated, then its space usage is proportionately reduced by 50%. The actual amount of memory pages that gets updated depends heavily on the DNN workload. The restore operation will transparently restore the most recent pages (i.e., parameters) of the model.

LATTICE offers the capability to maintain multiple persistent model versions, allowing users to access these past versions using the restore() LATTICE API call. This feature is useful for various tasks like model analysis and transfer learning. However, this simplistic approach to versioning brings forth a new challenge. The use of "eager versioning" can potentially lead to training stalls. In LATTICE, which relies on in-place updates rather than creating separate DNN snapshots iteratively, the memory pages under the *Checkpoint Directory* are set to read-only during the version creation process to ensure checkpoint consistency. Consequently, new checkpoint operations cannot commence until the "eager versioning" operation is completed. To address this challenge and optimize "eager versioning," LATTICE parallelizes the version creation process using OpenMP [30] - a technique that involves traversing and copying data and metadata structures of the persisted model concurrently using multiple threads.

5 Evaluation

In this section, we employ ML model training under different scenarios to assess the effectiveness of LATTICE in comparison to the prevailing model versioning techniques across several DNNs. Our evaluation seeks to address two primary questions. (1) How LATTICE's model versioning impacts the execution of the DNN training job? and (2) How much

space efficiency can LATTICE's in memory model versioning offer?

5.1 Experiment Setup

Table 3: Storage backend configuration (LVM=Logical Volume Manager).

Storage backend for experiments
SSD: 512GB Intel SSD Pro 7600p Series SSD (EXT4)
PM: 4x128GB Intel Optane NVDIMM (EXT4+LVM)

Table 4: DNN models evaluated.

ID	Name	Size(MB)	# of Layers
M1	Efficientnet_b0 [37]	14	135
M2	Densenet201 [18]	60	305
M3	Darknet53 [34]	155	77
M4	Alexnet [23]	223	14
M5	VGG-16 [35]	513	25

Servers. As shown in Table 3, for our comprehensive evaluation, we deployed a server configuration featuring dual Intel Xeon Gold 5218R CPUs, 128GB of DRAM, 4TB of SAS storage, and 2 Nvidia Quadro RTX 5000 GPUs. We use two storage backends for our experiments - SSD and Persistent Memory (PM). The choice of PM as evaluation platform is driven by multiple factors - (1) the commercial unavailability of low-cost CXL-attached byte-addressable NVM expansion devices, (2) emergence of CXL-attached PM in research domain [19], and (3) the evolution of the PM software stack on the operating system side for CXL-attached byte-addressable NVM expansion devices (i.e., CXL Type 3 devices).

We integrate LATTICE with a lightweight DNN training framework called Darknet [33] and deploy LATTICE's persistent containers on the PM backend using `fsdax`. LATTICE can be deployed on CXL-attached NVM expansion devices, when they become available, just by removing the PM flush operations in the library to accomodate for CXL's Global Persistent Flush (GPF) feature. The operating system transparently switches from PM device driver to CXL device driver in such a case. To compare LATTICE with traditional file I/O-based model versioning, we turn off CheckFreq's space optimization feature to stop removing older checkpoint files. CheckFreq is deployed on both SSD and PM backends, and denoted as CheckFreq (SSD) and CheckFreq (PM) in the experimental results, where applicable. We were unable to evaluate LibPM for these workloads since it does not support memory allocations greater than 4KB.

Baseline. Choosing a baseline to compare LATTICE is a challenging task. Since, it is an in-memory model versioning solution, the first instinct would be to compare it against other in-memory checkpointing solutions. However, the primary focus of LATTICE is space-efficient DNN model versioning, which is absent in existing in-memory solutions and extending these solutions to support model versioning is not trivial. On the other hand, state-of-the-art filesystem based checkpointing solution CheckFreq [28] can be easily extended to store multiple versions of the model. In addition, CheckFreq has similar space utilization requirements as existing in-memory checkpointing solutions[1]. Hence, we choose CheckFreq as the baseline in our evaluation. Approximately 900 lines of code has been added to Darknet to implement CheckFreq and integrate LATTICE.

Workloads. In our evaluation, as listed in Table 4 we utilized five well-known DNN image classification models to assess LATTICE (Table 4). These models vary in terms of model size and the number of layers. We use the CIFAR-10 dataset [22] to run the DNN training jobs.

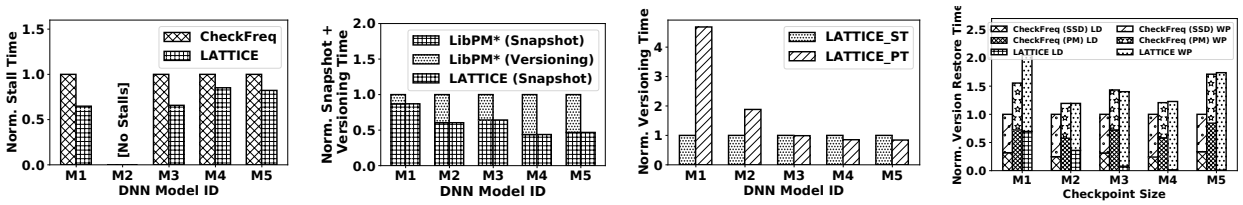
5.2 Evaluation Results

In this subsection, we examine the impact of low-latency direct persistence checkpointing on DNN training, evaluate the performance of two versioning techniques, and analyze the restore time using LATTICE.

5.2.1 Impact on Training. We examine LATTICE's impact on DNN training by analyzing its effect on training stall time. Reducing training stalls due to model versioning directly leads to reduced training time. To assess LATTICE's impact in the worst-case, we deploy it on a machine without the eADR feature (analogous to GPF in CXL), with DDIO enabled. This setup forces data transfers through the CPU cache, requiring explicit cache flush operations for persistence and incurring the highest snapshot overheads. We collect data for creating 100 model versions for each model. Despite the challenging setup, LATTICE reduces DNN training stall time between 18%-35% compared to file I/O-based DNN model versioning across four out of five models (Figure 4a). Notably, none of the methods incurred stall time for Densenet201[M2].

Next, we aim to understand LATTICE's versioning performance against other NVM-specific libraries. However, doing it directly is challenging, since none of the in-memory solutions supports model versioning and implementing versioning support in these solutions requires a lot of efforts. Incorporating the optimizations proposed by these libraries, such as GPM's snapshot mechanism without DDIO [31] and LightCheck's snapshot sharding [6], into LATTICE also requires nontrivial amount of work. Intuitively, if LATTICE

[1]In-memory checkpointing solutions usually maintain a small amount of metadata for pointer discovery. Such metadata are only a few KBs in size, which is insignificant compared to the hundreds of MBs used to store a model. [27, 31]



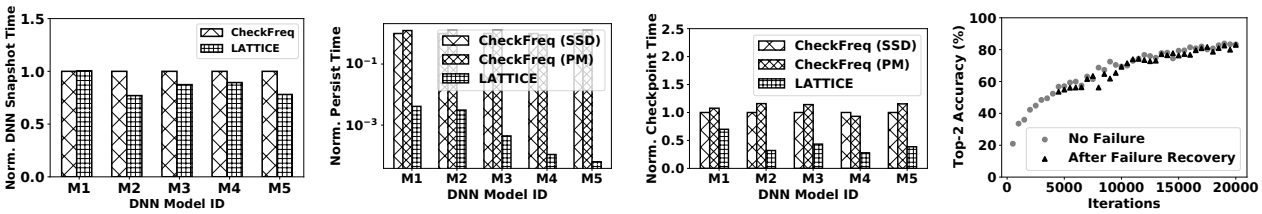
(a) (b) (c) (d)

Figure 4: LATTICE - (a) reduces checkpointing stall time up to 35%, compared to file I/O-based checkpointing, (b) is estimated to reduce chances of blocking foreground operation between 12%-53%, (c) can improve versioning time through parallelism for larger DNN models, (d) version restore time depends on number of layers in the network, and model size comes into play only when pushing the weights to GPU.

and GPM [31] uses the same GPU-NVM data transfer mechanisms (i.e., avoiding DDIO) and persist primitives (i.e., using fence for write ordering), then both of these solutions should have similar training stalls. Similarly, incorporating LightCheck's [6] snapshot sharding into LATTICE would reduce the stall time even further. If we were to estimate LATTICE's performance in comparison to LibPM, the difference would be significant. LATTICE is estimated to reduce the chances to a foreground operation being blocked by 12%-53%, compared to LibPM, by reorganizing the workflow (Figure 4b).

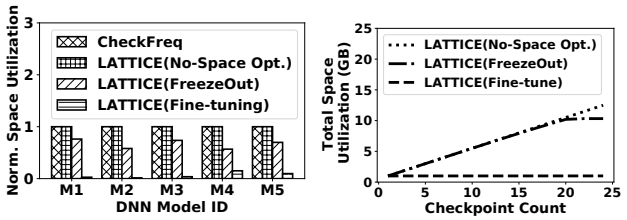
We also observe that versioning time in LATTICE is dependent on two primary factors - number of layers and number of parameters (i.e., memory pages) in each layer. If the number of layers in a DNN checkpoint is high, creating a version using parallel threads could be beneficial. However, if each layer contains relatively small number of memory pages on average, thread synchronization delay could outweigh the benefits of multiple threads. We observe this behavior in Figure 4c. In workloads, where checkpointing size is small (e.g. Efficientnet_b0 - 14MB), LATTICE_ST performs better than LATTICE_PT, because the overhead of fork-join operations for multiple threads to perform checkpointing is higher than the derived benefit if the checkpoint size is small (e.g. Efficientnet_b0 - 14MB). This behavior reverses as checkpoint size increases (e.g., VGG-16 - 513MB).

5.2.2 Space Efficiency. The amount of non-volatile memory pages occupied for storing data is highly dependent on the workload. We evaluate LATTICE in using three different workloads and compare its space-saving feature with CheckFreq. In the first workload (i.e., the No Space Optimization workload), all model parameters get updated between subsequent model versions, which provides no space optimization opportunity during version creation. This workload updates all memory pages in the working copy inside the LATTICE container during each checkpoint. The FreezeOut [5] workload progressively freezes the layers of the DNN, from top to bottom, after each version creation operation in LATTICE. The fine-tuning workload updates the last two layers of the network and versions the model inside the LATTICE container. We measure the total amount of space required to store the model versions, along with the space required for maintaining the working copy in LATTICE. Since CheckFreq does not provide any space optimization opportunity, it uses the same amount of space for all three workloads. Hence, we use only one bar to represent CheckFreq's space utilization in Figure 6a. As depicted in Figure 6a, LATTICE requires 23.8% to 43.2% less space to store the checkpoints for the FreezeOut, when model layers are pro. The difference in space saving comes from the size variance of top layers with bottom layers in a model. For example, the two connected layers at the bottom of VGG-16 network corresponds to approximately 70% of the checkpoint size. Hence, space saving is comparatively lower for VGG-16. On the other hand, Darknet53 and Efficientnet_b0 have lower variances in the top and bottom layer sizes, and space savings are comparatively higher for these models. We also see similar trends in space savings for the fine-tuning workload. However, the fine-tuning workload shows much higher space savings overall, compared to FreezeOut [5]. As depicted in Figure 6a, LATTICE saves 84.8% to 98.9% space when checkpointing only the fine-tuned layers. Figure 6b shows an increase in total space utilization to store model versions in LATTICE 1) when layers are frozen progressively and 2) when the last two layers of the VGG-16 model are fine-tuned, as training progresses. Although the space utilization of the fine-tuning workload for VGG-16 seems flat, in reality, the space utilization increases 76 KB with each checkpoint. In addition, the initial space utilization is higher for fine-tuning, since we version the full model at the beginning and also count the space required to maintain the working copy in LATTICE. The space utilization of FreezeOut workload closely follows the space utilization of the no space optimization workload till the 20th version (i.e., until the 20th layer is frozen). After this version, the



(a) (b) (c) (d)

Figure 5: LATTICE (a) reduces DNN snapshot time up to 23% of file I/O-based checkpointing, regardless of storage media for the latter, (b) ensures early persistence at the fraction of time compared to file I/O-based checkpointing on SSD and PM, (c) reduces checkpoint time up to 72% of file I/O -based checkpointing on SSD, (d) preserves accuracy of the DNN model (Alexnet) after restart (at 4000th iteration).



(a) (b)

Figure 6: LATTICE - (a) reduces space utilization for versioned checkpoints for different workloads, and (b) cuts down space utilization by 92% for VGG-16 fine-tuning workloads, where only the last 2 layers are being tuned.

fully connected layers, which hold most of the parameters in VGG-16, are frozen and the space utilization reduces sharply.

5.2.3 Other Performance Metrics. Next, we evaluate LATTICE's performance for version checkpointing tasks during regular training. We measure LATTICE's DNN snapshot (GPU-PM copy) latency and compare it with CheckFreq's CPU-snapshot (memory allocation + GPU-DRAM copy) latency. CheckFreq copies data from GPU HBM to CPU DRAM during CPU-snapshot, regardless of the underlying storage media used for writing checkpoints. Figure 5a shows that LATTICE can reduce DNN snapshot latency by up to 23%, compared to CheckFreq. This is because LATTICE eliminates repeated memory allocations for every snapshot from the critical path and performs in-place updates of model parameters inside the LATTICE container.

Next, we evaluate LATTICE's persist latency (Figure 5b). LATTICE can reduce persist latency up to 99.99% compared to the file I/O-based checkpointing method employed by CheckFreq with either SSD or persistent memory as storage media. LATTICE achieves this by bypassing the overheads of the file I/O-based storage stack and leveraging direct persistence on persistent memory devices instead. Finally, we compare the actual checkpointing/versioning latency realized by LATTICE in Figure 5c. LATTICE's actual checkpointing/versioning latency is up to 72% lower compared to CheckFreq's best performing variant (SSD). It is not hard to deduce that the actual checkpointing time of in-memory solutions such as GPM, LightCheck and Portus are going to be lower than LATTICE, since these solutions do not need to execute version creation operations. Theoretically, the worst-case actual checkpointing/versioning latency in LATTICE would be a $\leq$ 2x of in-memory checkpointing latency without versioning, when all optimizations are applied equivalently. LATTICE trades off versioning latency with space efficiency. Hence, in applications where space-efficient in-memory model versioning is not needed and more frequent checkpoint bears higher importance, existing in-memory checkpointing operations may be preferable over LATTICE.

We also check the correctness of LATTICE's checkpointing mechanism. In this experiment, we train Alexnet for 20000 iterations without any failure and measure its validation accuracy periodically. Then we restart training (to simulate a failure recovery) using a checkpoint taken at the 4000th iteration and train the model again till 20000 iterations. From Figure 5d, we see how LATTICE preserves the accuracy of the model even after restarting at the 4000th iteration. This is a crucial metric for checkpointing systems, as maintaining model accuracy during the training process is of utmost importance.

Lastly, we measure LATTICE's version restore time. The version restore process consists of two steps - loading (LD) the version from container or file in CPU DRAM, and copying the model parameters back to GPU. We observe that LATTICE has higher restore time compared to file I/O-based checkpointing methods (Figure 4d). This is attributed to relatively higher data movement time in LATTICE during restore operation. Asynchronous GPU-NVM and parallel NVM-NVM data copy operations, used to hide the latency of the

version creation, cannot be used during the restore process to hide restore latency. This can be improved by turning off DDIO during restore [31]. However, efforts to reduce NVM-GPU copy time are unlikely to result in significant performance improvements. Even with failure-prone training sessions, restore operations occur significantly less often than the number of times checkpoints are triggered during training.

6 Related Works

Existing reseacrh on DNN model versioning can be divided into two groups. First, DNN model management systems with a focus on managing ML lifecycle. Second, traditional file I/O-based model checkpointing systems that create multiple versions of the model.

The model version management systems are focused on maintaining ML lifecycle for an organization. ModelDB [39] uses different techniques to store, track and index large number of models with enough context to be queried and discovered by other users in the organization. MLFlow [45] is focused on keeping track of experimental inputs (e.g., data versions, code and hyperparameters), reproducibility of results and deployment of models in production. Mystique [38] uses quantization, summarization and data de-duplication techniques to optimize space efficiency for storing large number of models. LATTICE is orthogonal to these solutions and can be integrated as a backend library in these systems.

Traditional checkpointing solutions that store multiple checkpoint versions usually focuses on training stalls and/or space efficiency for a single checkpoint. Such solutions include use of quantization or compression to reduce the size of each model version separately [1, 7]. Such techniques need additional resources and may not always provide significant benefits in increasing space efficiency size [13]. Check-N-Run [13] demonstrates the use of differential checkpointing, which drops stale vectors from a checkpoint to reduce size. Since it is the deep learning application that identifies and omits stale vectors from the checkpointing in the case of Check-N-Run, this technique can also be used in combination with LATTICE. The other set of checkpointing solutions focuses on masking checkpointing delay using a variety of techniques. DeepFreeze [29] uses an asynchronous checkpointing method to mask delay, along with efficient data serialization and sharding. However, masking checkpointing delay can still lead to DNN snapshotting stalls [47]. GPM [31] demonstrates how to minimize training stalls using byte-addressable non-volatile memory. LightCheck [6] optimizes the above through layer-wise checkpointing, and Portus [24] uses `devdax` with a RDMA datapath to optimize checkpoiting in distributed training. Although these solutions can't be extended to support space-efficient versioning easily due to double buffering, other optimizations used in these solutions can be integrated with LATTICE.

7 Discussions

Non-Volatile Memory Capacity and Cost Concerns. LATTICE incorporates an incremental versioning approach. However, storing read-only versions without deletion in LATTICE can strain storage resources. In particular, with highly frequent versioning at low intervals, the storage capacity on a single device could be exhausted quickly. The most obvious strategy to over the capacity concern is to add more non-volatile memory devices in the system. While this can increase the cost of the system, such cost increases are likely to be a small fraction of the overall system cost, where state-of-the-art GPUs incur most of the capital investment[2]. While cost concern is unlikely to throttle capacity increase of non-volatile memory in high-end DNN training systems, limited availability of PCIe memory slots and higher latency of accessing remote CXL-attached memory can effectively limit the capacity of such devices in a system. Hence, other strategies are needed to address the capacity bottleneck of non-volatile memory devices. The capacity bottleneck can be effectively resolved with LATTICE's container-based design, which is portable across devices. If the DL application aims to preserve all model versions, it can initialize a new LATTICE container each time an existing one reaches its capacity limit and move the old container from persistent memory to other persistent storage systems, such as networked storage, file systems, or object stores. If only specific model versions need to be preserved, an application can transfer (i.e., restore to CPU DRAM and model version in a new container in the background) selected model versions from the old LATTICE container to a new container when it approaches its available capacity, and then delete the old container. It is also possible to write a small application to port in-memory checkpoints from a LATTICE container to the filesystem, similar to Portus [24].

LATTICE integration and support for versioning beyond DL training. LATTICE provides a low-level API in C that can be integrated inside the checkpointing module of different DNN training frameworks to provide transparent in-memory model versioning. To optimize space utilization during versioning, it is best for the ML application to control annotation and copying of modified data from GPU to NVM, as demonstrated by Check-N-Run [13]. LATTICE can also be used with complex training algorithms, such as model-parallel or pipeline-parallel training. Existing in-memory checkpointing solutions, such as Portus [24] and LightCheck

[2]When we compare pricing information of NVIDIA DGX systems and Intel Optane Persistent Memory devices (before the latter was discontinued), adding 512GB persistent memory module results in $< 1\%$ cost increase.

[6], can work as a reference for such integration. LATTICE, with its efficient and low-latency versioning capabilities, can also extend its support beyond deep learning (DL) training to various other applications and use cases. For instance, in production environments where multiple model versions are deployed for inference, LATTICE can be used to persist model versions in memory. By creating versions at different stages of model development, organizations gain the ability to seamlessly roll back to prior model versions if issues or disruptions arise with the latest deployment. DL model versioning support provided by LATTICE encourages a culture of continuous improvement in deep learning model deployments. It allows organizations to experiment with new model versions, fine-tune them, and collect feedback from real-world usage. If a newer version does not perform as expected, the system can revert to a known and reliable previous version while issues are addressed.

Limitations and Future Works. The LATTICE container abstraction is deployed on top of `fsdax` I/O interface. This makes the container easily manageable and portable. The `fsdax` I/O stack enables LATTICE create and manage the containers as files on top of a `dax`-enabled filesystem. However, the `fsdax` I/O stack is slower than the `devdax` I/O stack. In the future, we plan to adapt LATTICE for `devdax`. This would require us to adopt the page allocation algorithm for `devdax`. The current implementation of LATTICE does not support the deletion, branching, or merging of specific model versions. The root data structure in LATTICE can be extended to support the above mentioned features with the addition of a version dependency management component. We plan to develop these features in the future. In addition, LATTICE only supports a single *root* object inside a container. Hence, a single container cannot be used to store versions of different DNN models. Although multiple containers can be used to solve this issue, support for versions of different DNN models will be added in the future. We also plan to optimize the version restore time of LATTICE for emerging applications by integrating smart prefetching techniques. LATTICE does not have any inherent conflict with model-parallelism or pipeline-parallelism. It is an engineering task to integrate LATTICE in such training algorithms using RDMA (for CXL no additional implementation is needed). Existing in-memory checkpointing solutions, Portus and LightCheck, can serve as a reference for such implementations.

8 Conclusion

This paper introduces LATTICE, a persistent container-based DNN model versioning library optimized for DNN workflows. LATTICE eliminates redundancies, reduces stalls, and streamlines operations to enhance versioning efficiency. Its proactive approach, with eager versioning and early persistence, minimizes versioning time while ensuring fault tolerance. Extensive evaluations show that LATTICE reduces persistence time by up to 99.99%, end-to-end versioning time by up to 72%, and runtime overheads by up to 35%, compared to file I/O-based checkpointing and versioning solutions. Furthermore, it increases versioning frequency by 20%–384% for large models, outperforming file I/O-based methods. LATTICE also reduces space utilization during version creation for different workloads. LATTICE's space utilization reduction for FreezeOut workloads, where model layers are frozen progressively from top to bottom, ranges from 23.8% to 43.2%, compared to other checkpointing solutions. For fine-tuning workloads where only the last two layers are being tuned, LATTICE can reduce NVM space utilization from 84.8% to 98.9%.

Acknowledgments

We thank the reviewers and our shepherd, Antonio Barbalace, for their constructive comments that have significantly improved the paper. This work was supported by the following NSF grants: CSR-2402328, CAREER-2338457, CSR-2406069, CSR-2323100, HRD-2225201, CCF-1718335, CNS-1956229, IIS-2331908, and gifts from Seagate and NetApp.

References

[1] [n. d.]. ZIP (file format). https://en.wikipedia.org/wiki/ZIP_(file_format).

[2] Alexandro Baldassin, João Barreto, Daniel Castro, and Paolo Romano. 2021. Persistent Memory: A Survey of Programming Support and Implementations. *ACM Comput. Surv.* 54, 7, Article 152 (jul 2021), 37 pages. doi:10.1145/3465402

[3] Lorena Barreto Simedo Pacheco, Musfiqur Rahman, Fazle Rabbi, Pouya Fathollahzadeh, Ahmad Abdellatif, Emad Shihab, Tse-Hsun (Peter) Chen, Jinqiu Yang, and Ying Zou. 2024. DVC in Open Source ML-development: The Action and the Reaction. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI* (Lisbon, Portugal) *(CAIN '24)*. Association for Computing Machinery, New York, NY, USA, 75–80. doi:10.1145/3644815.3644965

[4] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. 2017. Freezeout: Accelerate training by progressively freezing layers. *arXiv preprint arXiv:1706.04983* (2017).

[5] Andrew Brock, Theodore Lim, J. M. Ritchie, and Nick Weston. 2017. FreezeOut: Accelerate Training by Progressively Freezing Layers. arXiv:1706.04983 [stat.ML] https://arxiv.org/abs/1706.04983

[6] Menglei Chen, Yu Hua, Rong Bai, and Jianming Huang. 2023. A Cost-Efficient Failure-Tolerant Scheme for Distributed DNN Training. *Proceedings of the 41st IEEE International Conference on Computer Design (ICCD)* (2023).

[7] Yu Chen, Zhenming Liu, Bin Ren, and Xin Jin. 2020. On Efficient Constructions of Checkpoints. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. JMLR.org, Article 152, 10 pages.

[8] Joel Coburn, Adrian Caulfield, Ameen Akel, Laura Grupp, Rajesh Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making

Persistent Objects Fast and Safe With Next-Generation, Non-Volatile Memories. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*.

[9] CXL Consortium. [n. d.]. Compute Express Link. https://computeexpresslink.org/

[10] Intel Corporation. [n. d.]. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

[11] Catello Di Martino, Zbigniew Kalbarczyk, Ravishankar K. Iyer, Fabio Baccanico, Joseph Fullop, and William Kramer. 2014. Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 610–621. doi:10.1109/DSN.2014.62

[12] Wenxin Ding, Arjun Nitin Bhagoji, Ben Y. Zhao, and Haitao Zheng. 2024. Towards Scalable and Robust Model Versioning. In *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. 592–611. doi:10.1109/SaTML59370.2024.00036

[13] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Mikhail Smelyanskiy, and Murali Annavaram. 2020. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *Symposium on Networked Systems Design and Implementation*.

[14] GitHub. [n. d.]. Git large File Storage. https://git-lfs.com/

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[16] Gao Huang, Yixuan Li, Geoff Pleiss, Zhuang Liu, John E. Hopcroft, and Kilian Q. Weinberger. 2017. Snapshot Ensembles: Train 1, Get M for Free. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJYwwY9ll

[17] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[18] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[19] SK Hynix. [n. d.]. How Selector-Only Memory Emerged as the Leading Solution for CXL. https://news.skhynix.com/how-selector-only-memory-emerged-as-the-leading-solution-for-cxl/

[20] Hongpeng Jin, Wenqi Wei, Xuyu Wang, Wenbin Zhang, and Yanzhao Wu. 2023. Rethinking Learning Rate Tuning in the Era of Large Language Models. arXiv:2309.08859 [cs.LG]

[21] R. Madhava Krishnan, Diyu Zhou, Wook-Hee Kim, Sudarsun Kannan, Sanidhya Kashyap, and Changwoo Min. 2023. TENET: Memory Safe and Fault Tolerant Persistent Transactional Memory. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 247–264. https://www.usenix.org/conference/fast23/presentation/krishnan

[22] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[24] Yuanhao Li, Tianyuan Wu, Guancheng Li, Yanjie Song, and Shu Yin. 2024. Portus: Efficient DNN Checkpointing to Persistent Memory with Zero-Copy. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. 59–70. doi:10.1109/ICDCS60910.2024.00015

[25] Xiaodong Ma and Weiwei Fang. 2023. Deep Reinforcement Learning Based Multi-Task Automated Channel Pruning for DNNs. In *2023 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–9.

[26] Kiwan Maeng, Shivam Bharuka, Isabel Gao, Mark Jeffrey, Vikram Saraph, Bor-Yiing Su, Caroline Trippel, Jiyan Yang, Mike Rabbat, Brandon Lucia, and Carole-Jean Wu. 2021. Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 637–651. https://proceedings.mlsys.org/paper_files/paper/2021/file/f0f9e98bc2e2f0abc3e315eaa0d808fc-Paper.pdf

[27] Leonardo Marmol, Mohammad Chowdhury, and Raju Rangaswami. 2018. LibPM: Simplifying Application Usage of Persistent Memory. *ACM Trans. Storage* 14, 4, Article 34 (dec 2018), 18 pages. doi:10.1145/3278141

[28] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *USENIX Conference on File and Storage Technologies*.

[29] 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

[30] OpenMP. [n. d.]. https://www.openmp.org/.

[31] Shweta Pandey, Aditya K Kamath, and Arkaprava Basu. 2022. GPM: Leveraging Persistent Memory from a GPU. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 142–156. doi:10.1145/3503222.3507758

[32] Pmem.io. [n. d.]. Persistent Memory Programming. http://pmem.io.

[33] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[34] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An Incremental Improvement. *arXiv e-prints*, Article arXiv:1804.02767 (April 2018), arXiv:1804.02767 pages. doi:10.48550/arXiv.1804.02767 arXiv:1804.02767 [cs.CV]

[35] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*.

[36] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

[37] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6105–6114. https://proceedings.mlr.press/v97/tan19a.html

[38] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1285–1300. doi:10.1145/3183713.3196934

[39] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. ModelDB: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data*

Analytics (San Francisco, California) *(HILDA '16)*. Association for Computing Machinery, New York, NY, USA, Article 14, 3 pages. doi:10.1145/2939502.2939516

[40] Haris Volos, Andres Jaan Tack, and Michael Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*.

[41] Feng Wang, Guoyizhe Wei, Qiao Liu, Jinxiang Ou, xian wei, and Hairong Lv. 2021. Boost Neural Networks by Checkpoints. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 19719–19729. https://proceedings.neurips.cc/paper_files/paper/2021/file/a40511cad8383e5ae8ddd8b855d135da-Paper.pdf

[42] 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

[43] P William, Ritik Gade, Rup esh Chaudhari, AB Pawar, and MA Jawale. 2022. Machine learning based automatic hate speech recognition system. In *2022 International conference on sustainable computing and data communication systems (ICSCDS)*. IEEE, 315–318.

[44] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. 2019. A Survey on Neural Architecture Search. arXiv:1905.01392 [cs.LG]

[45] Matei A. Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. 2018. Accelerating the Machine Learning Lifecycle with MLflow. *IEEE Data Eng. Bull.* 41 (2018), 39–45. https://api.semanticscholar.org/CorpusID:83459546

[46] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. arXiv:2306.05685 [cs.CL]

[47] Yuchen Zhong, Guangming Sheng, Juncheng Liu, Jinhui Yuan, and Chuan Wu. 2023. SWIFT: Expedited Failure Recovery for Large-scale DNN Training. arXiv:2302.06173 [cs.DC]

[48] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2021. A Comprehensive Survey on Transfer Learning. *Proc. IEEE* 109, 1 (2021), 43–76. doi:10.1109/JPROC.2020.3004555

Received 26 May 2025; revised 4 August 2025