FitCache: A Transparent Drop-In Framework for Multi-Tier Caching to Accelerate Distributed Deep Learning Workloads

Guangxing Hu
North Carolina State University
Raleigh, USA
ghu4@ncsu.edu

Awais Khan
Oak Ridge National Laboratory
Oak Ridge, USA
khana@ornl.gov

Christopher Zimmer
Oak Ridge National Laboratory
Oak Ridge, USA
zimmercj@ornl.gov

Michael J. Brim
Oak Ridge National Laboratory
Oak Ridge, USA
brimmj@ornl.gov

Frank Mueller
North Carolina State University
Raleigh, USA
fmuelle@ncsu.edu

***Abstract*—Training in Deep learning (DL) remains highly compute- and data-intensive, with I/O becoming a critical bottleneck as models and datasets scale. Recent studies report that data loading can dominate training time, especially on large-scale HPC systems with shared parallel file systems (PFS). Existing caching approaches either rely on single-tier designs or require intrusive modifications to training pipelines, limiting their portability and effectiveness. In this work, we present FitCache, a transparent drop-in framework for multi-tier caching to accelerate distributed DL training by coordinating fast local memory (*e.g.*, DRAM, Persistent Memory (PMem)) and NVMe as hierarchical caches atop PFS. Our design adapts to hardware diversity, *i.e.*, if NVMe is missing, memory transparently acts as a caching tier, ensuring stable performance. FitCache transparently intercepts I/O requests and issues concurrent fetches across all tiers, returning data from the fastest responder without centralized metadata or static redirection paths. FitCache adapts to dynamic workloads and heterogeneous clusters while maintaining POSIX compatibility. Experiments on Frontier (2048 GPUs) and smaller research clusters show that FitCache reduces training time by up to 40% and per-batch I/O latency by up to 71.6% compared to Lustre Orion PFS, offering a drop-in solution for scalable DL training.**

***Index Terms*—High-Performance Computing (HPC), Deep Learning, Memory, Caching and I/O Optimizations**

I. Introduction

Deep learning (DL) has become a dominant paradigm across a wide range of domains, from healthcare [1], [2], climate science [3], smart cities [4], [5] to NLP [6]–[9] and computer vision [10], [11], driving both practical applications and foundational advances. The global DL market is projected to surpass 41.39 billion by 2030 [12]. However, DL training remains compute- and data-intensive [13], demanding significant resources for data loading, gradient computation, and checkpointing [14].

Distributed DL typically uses three strategies: data parallelism [15], model parallelism [16], and pipeline parallelism [17]. On large-scale clusters like Frontier [18], data parallelism is the most common. This approach uses many compute accelerators such as GPUs, FPGAs, or custom ASICs, to scale DL training vertically and horizontally. Each accelerator holds a full copy of the model and processes a subset of the training data in parallel. Many studies focus on improve compute efficiency [19], job scheduling [20], and data communication [21] in this setting. This is because data-parallel training requires both high compute performance and networking throughput, often with many GPUs, and frequent communication [22]. After each step, the new gradients are consolidated via an all-reduce or all-gather, which involves all GPUs, to update the model.

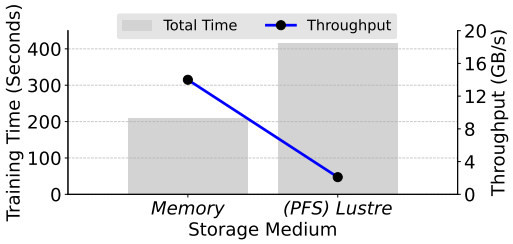


Fig. 1: Per-epoch training time and memory vs. Lustre throughput during a DeepCAM job on 64 AMD Mi250X GPUs.

A general challenge with DL training is posed by large training sets, which is the focus of this work. Recent studies report that I/O can account for up to 80% of total training time in DL workloads [23], [24], highlighting the critical role of efficient data storage and retrieval in determining end-to-end training performance. To quantify the impact of storage configuration on distributed DL performance, we conduct a controlled experiment comparing memory and parallel file systems (PFS). As shown in Figure 1, using Lustre [25] PFS increases training time by up to 2× compared to fast local memory (*e.g.*, DRAM), despite identical training configurations. Yet, training sets are too large to fit into memory.

As modern accelerators process training samples at high throughput, storage performance becomes a critical factor in both end-to-end training time and GPU utilization, directly impacting overall system efficiency. As computational throughput increases, data movement struggles to keep pace. For example, NVIDIA introduced tensor cores, which speed

up training using FP16 and INT8 instructions on recent GPUs. Google's TPU pods scale training across up to one thousand TPUs. These high-throughput systems frequently stall waiting for data, with all accelerators blocked by the slowest batch. To address these challenges, recent work [26]–[31] explores storage and memory-side optimizations that reduce PFS congestion and improve data locality, thereby accelerating both checkpointing and training pipelines. However, existing approaches often assume a fixed caching architecture or focus on single-tier optimizations, overlooking the heterogeneity of modern HPC memory and storage hierarchies. For example, frameworks like HVAC [30] treat node-local NVMe as the sole cache tier, ignoring faster and underutilized memory. These designs fall short on large-scale supercomputers like Frontier, where training datasets span terabytes and must be streamed efficiently to thousands of GPUs. They also fail to generalize across platforms with different hardware configurations.

Beyond DL-specific caching systems, several general-purpose memory-based caching frameworks [32]–[35] have been proposed to accelerate data access. However, these frameworks are not designed for high-performance training pipelines and exhibit compatibility and scalability challenges in HPC environments. In-memory key-value stores such as Redis [32] and Memcached [33] provide fast data access but lack POSIX file system semantics. As a result, they are not compatible with unmodified DL training code. In contrast, multi-tier caching frameworks, such as Hermes [34], use multiple storage tiers for data caching, but they add cross-tier file residency tracking overhead. Kangaroo [35] introduces eviction and indexing mechanisms to balance DRAM index cost and flash write amplification. These designs incur runtime overheads and often fail to adapt to dynamic workloads or heterogeneous hardware configurations, limiting their applicability on large-scale HPC systems.

Despite these efforts, several challenges remain open in large-scale distributed DL training. First, DL applications typically rely on file system semantics and read training data through POSIX interfaces. While storage hierarchies in modern supercomputers include HBM, DRAM, Persistent Memory (PMem) [36], NVMe, and PFS, existing systems fail to leverage these tiers holistically. They either treat them in isolation or require manual configuration. Second, most systems rely on static redirection paths or centralized metadata services and lack runtime mechanisms to select the fastest available tier. This results in unnecessary delays and inefficient bandwidth utilization, especially under runtime variability such as load imbalance across nodes or device-level performance variance. Third, these systems are typically tailored to fixed hardware setups and do not generalize to heterogeneous clusters, i.e., in memory-rich systems (*e.g.*, Andes [37]), storage-supplemented compute nodes (*e.g.*, Frontier), or hybrid memory nodes with PMem. Finally, many approaches break compatibility with existing DL code, requiring changes to data pipelines or application logic, which limits adoption in practice.

To address these challenges, we propose FITCACHE, a portable and transparent framework for multi-tier file caching in distributed DL workloads. Our approach adapts to available hardware by combining fast local memory (*e.g.*, DRAM and PMem) and NVMe into a coordinated multi-level cache on both small- and large-scale clusters. FITCACHE generalizes to arbitrary storage hierarchies, *i.e.*, additional tiers can be added as needed. FITCACHE treats the addition or absence of a storage tier not as a limitation but as an opportunity for dynamic adaptation. This design enables efficient I/O acceleration across heterogeneous HPC systems, from supercomputers to smaller research clusters with zero-modifications to the DL training code or infrastructure.

To further reduce I/O latency, we introduce an interposing mechanism for multi-tier fetches: When a file is first requested, an I/O request is intercepted so that our runtime launches parallel I/O requests to all relevant tiers (*e.g.*, memory and storage) and returns the result from the fastest responder. This interposing strategy avoids central metadata bottlenecks and hides device-level variance. And we use a tier-aware caching policy that prioritizes hot data to be delegated to the fastest available memory, which aims at high utilization of faster tiers combined with lower latencies and higher bandwidth.

In summary, this paper makes the following contributions:

- To address the I/O bottlenecks in distributed DL, we propose FITCACHE, a portable and transparent framework for multi-tier file caching that accelerates training by intelligently leveraging heterogeneous memory and storage resources. Our system is designed to operate efficiently across HPC environments with varying hardware capabilities.
- To minimize latency, we develop a concurrent I/O requests strategy to multiple tiers and promote the fastest response, eliminating the need for strict metadata tracking and tolerating tier variability. Furthermore, a cache placement policy adapts to available capacity, prioritizing frequently reused data for fast memory while spilling overflow to slower storage tiers.
- Experiments on both large- and small-scale clusters show that FITCACHE reduces training time by up to 40% on Frontier using 2048 GPUs and more than 30% on smaller clusters. It also cuts per-batch I/O time by up to 71.6% on 1024 GPUs compared to Lustre Orion PFS, significantly improving end-to-end training throughput.

II. BACKGROUND AND MOTIVATION

A. Distributed DL and the I/O Bottleneck

Popular ML frameworks [38], [39] represent Distributed DL computation as a data-flow graph. Graph edges carry data as multi-dimensional tensors. Graph nodes run operators such as matrix multiplication that turn inputs into outputs. One training iteration has three steps. The engine forwards a data batch to compute loss, runs a backward pass to get gradients, and updates the weights. The whole DL training process will repeat many iterations. Model developers design the graph. The execution engine then optimizes it and runs it on the device. When the model or dataset becomes too large for one

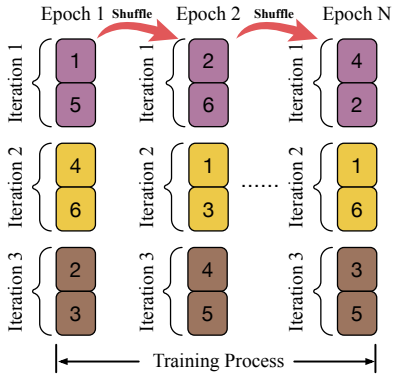


Fig. 2: Access pattern of a training dataset with 6 files and a batch size of 2. Each number shows a file ID. Files are shuffled and read in random order at the start of each epoch.

device to finish training within a reasonable time, we switch to distributed DL, *e.g.*, data parallelism, model parallelism, and pipeline parallelism. Those methods split the computation and run it on many distributed devices in parallel. When these stages run in parallel across different batches, the job is GPU-bound as the accelerators stay busy.

On production HPC systems, however, storage stalls frequently diminish effective parallelism, leaving GPUs idle and rendering the job I/O-bound. When they must share results, such as gradients, they perform an all-gather or all-reduce, typically via MPI [40], RCCL [41] or NVIDIA NCCL [42]. Because all GPUs must wait for the slowest batch, any storage stalls are amplified as we scale out, making I/O a dominant bottleneck. This causes data loading from storage to GPUs to become a major performance bottleneck for distributed DL training. Yet at petascale and beyond, the storage-to-GPU data path fails to scale proportionally. Former research shows that input loading stalls can account for more than 65-80% of iteration time [23], [24]. Eliminating this I/O bottleneck is therefore critical for efficient distributed training.

Distributed DL Training Pipeline: We analyze common access patterns and I/O characteristics in DL workloads. During training, each epoch scans the full dataset from the shared filesystem. The framework shuffles the access order before each epoch, but it does not change the set of files. For simplicity, we illustrate the access pattern using a small dataset with six files, as shown in Figure 2. Before training begins, the batch size is set. This defines how many files each compute unit reads at once. In this example, the batch size is 2. The dataset is therefore read in three batches during each training epoch. An epoch refers to one complete pass over the dataset. DL models usually require multiple epochs to converge. Before each epoch, the training framework shuffles the dataset to generate a new random file order. This randomization helps improve prediction accuracy and reduces overfitting. As a result, DL training shows the following access behavior: 1) The dataset is read-only during each epoch. 2) All files are read once per epoch and accessed again in later epochs. 3) The order of files changes across epochs to avoid overfitting. This reuse makes the workload cache-friendly [30], as DL jobs access the same data across epochs despite shuffling the file order for generalization.

B. Case Study: I/O Bottleneck in CosmoFlow on Frontier

While prior work and our analysis highlight the growing cost of I/O in distributed DL, we further quantify its practical impact through real-world profiling on a leadership-class system. We profile CosmoFlow [43], an HPC-optimized DL application for cosmology, on the Frontier supercomputer. We use the TensorFlow Profiler [44] to analyze the breakdown of each training batch. The profiling results, as detailed in Section IV-E, show that 64.08% of each batch is spent on I/O, and 60.06% of that time is attributed to loading data from Lustre Orion PFS. In contrast, GPU computation accounts for only 32.66%. After using one of the node-local storage cache systems [30], the I/O overhead accounts for nearly half of the total batch time. This underlines that I/O dominates the entire batch time. Since DL jobs reuse the same data samples across epochs, caching those samples after the first pass speeds up later epochs. While the data is reused, each epoch reshuffles the file access order to avoid temporal bias and improve generalization. If the cache cannot hold the full dataset, those random accesses cause misses and slow down the job. Fortunately, since individual samples are interchangeable from a caching perspective, the system may treat them independently of their order for staging purposes. Therefore, we propose a framework that exploits this reuse to streamline the I/O pipeline and reduce total training time.

C. Memory-Tiered Data Acceleration

Modern HPC systems such as El Capitan, Frontier, and Aurora offer vast compute and memory resources co-located within each node. Yet prior studies [45]–[47] reveal significant under-utilization: Peng et al. [46] report that over 90% of jobs consume less than 15% of available node memory, and memory usage stays below 35% for 90% of the runtime. This idle capacity presents an opportunity to repurpose memory for performance-enhancing strategies, such as in-memory caching.

In-memory caching offers a promising solution by staging frequently accessed data in fast local memory. Compared to traditional storage, memory technologies like DRAM and PMem provide orders-of-magnitude lower latency and higher bandwidth, enabling faster data access. Prior work [48]–[51] shows that memory-tiered caching can reduce data stalls and improve end-to-end training throughput. However, many of these methods lack application transparency or rely on custom APIs (*e.g.*, CoorDL [49]), limiting their adoption in production HPC settings. Others target only node-local memory, without a scalable, multi-node design that handles contention effectively. To overcome these scalability limitations, recent efforts have begun exploring distributed caching architectures that pool memory across nodes. Emerging work investigated CXL-enabled memory to construct distributed caching layers or even bypass GPU data paths in inference workloads [52]–

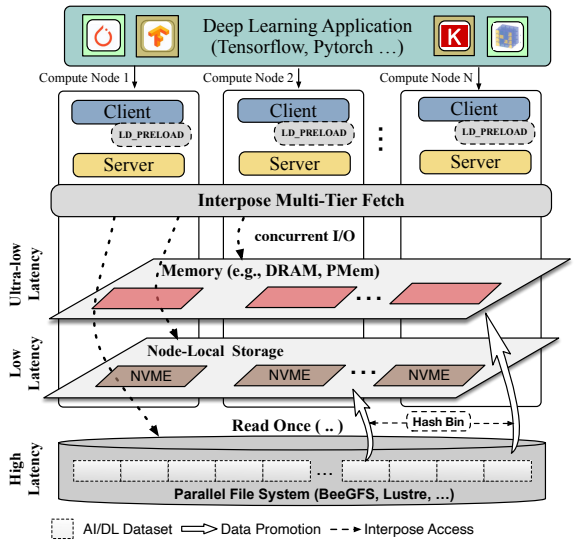


Fig. 3: System overview. A preload-based client redirects I/O to a server, which reads from multiple tiers and returns the fastest result.

[54], but these approaches remain early-stage and often require specialized hardware or runtimes.

D. Limitations of Existing Systems

a) ***Portable and Transparent Cache Systems:*** While many caching frameworks improve I/O performance, they often suffer from poor portability and limited transparency. Some are tailored to specific workloads (*e.g.*, Python-based DL pipelines [55]), while others require invasive changes to training code or data loading APIs [49]. While in-memory key-value stores offer high-speed access, their lack of POSIX-compliant interfaces and filesystem semantics makes them unsuitable as a transparent drop-in solution for existing applications. These constraints hinder adoption in production HPC environments, which run diverse workloads across various programming languages and frameworks. Moreover, several systems assume homogeneous hardware configurations and are tightly coupled to particular memory/storage hierarchies. This makes them impractical for deployment across heterogeneous clusters, such as those combining DRAM, PMem, NVMe, or other storage media. The lack of hardware-adaptive logic prevents these systems from efficiently utilizing storage and memory resources in the whole cluster, especially in environments with varying cache tier capacities. Finally, many prior systems rely on centralized metadata services to track cache status. This centralization introduces bottlenecks at scale and increases deployment complexity. In contrast, a portable cache system should integrate transparently into existing workflows, adapt to diverse hardware tiers, and avoid centralized dependencies. These goals motivate our design.

b) ***Efficient Data Retrieval on Multi-Cache Tier:*** Leveraging multiple cache tiers complicates data retrieval, since the system often does not know a sample's current location [56]. Prior evictions and shifting cache pressure may move data across tiers, introducing lookup overhead and increasing access latency if not properly managed. Existing methods often rely on centralized metadata services to track file residency, but this introduces synchronization overhead and limits scalability. Some use fixed redirection paths, assuming stable device performance across time and nodes. However, these static strategies cannot tolerate performance variability, such as temporary device slowdowns or imbalance across nodes, leading to frequent delays or degraded throughput. Moreover, centralized tracking imposes a single point of failure and limits deployment flexibility, especially in heterogeneous environments where tier availability varies across nodes. These limitations motivate the need for decentralized, adaptive mechanisms that can tolerate dynamic tier behavior without requiring global metadata or rigid access paths.

TABLE I: Comparison of FITCACHE with existing systems for distributed DL.

System	Tiered Systems	POSIX Compliance	Zero Code Changes	Low Overhead	Parallel Fetch
HVAC [30]	× (NVMe)	✓	✓	✓	×
CoorDL [49]	–	× (API)	×	–	×
Hermes [34]	✓	✓	✓	×	×
DIESEL [57]	×	✓	×	✓	×
SHADE [29]	× (In-mem)	× (API)	×	✓	×
DeepFetch [58]	×	× (API)	×	✓	×
Quiver [59]	×	× (API)	×	✓	×
NoPFS [60]	✓	×	×	✓	×
FanStore [61]	×	✓	✓	✓	×
FITCACHE	✓	✓	✓	✓	✓

c) ***Scalable Deployment on Leadership-Class Supercomputers:*** Although prior caching systems show promising results in small-scale or controlled settings, they fall short in real HPC deployments. First, most are tested only on a few nodes and have not been validated at supercomputer scale, leaving their scalability and robustness unproven. Second, some of the caching frameworks lack cross-node coordination; they operate only on local storage without sharing cache status across ranks. These limitations motivate the design of FITCACHE, a scalable and portable caching framework tailored for large-scale HPC environments. We deploy FITCACHE on both a research cluster with 80 compute nodes and the world's first exascale supercomputer with 9,402 nodes. This deployment demonstrates its ability to scale from tens to thousands of GPUs without code modification. FITCACHE supports cross-node caching through decentralized coordination, adapts to node-level resource variability, and integrates transparently with standard DL frameworks like TensorFlow and PyTorch. By removing the need for external metadata services or hardware-specific tuning, FITCACHE offers a practical path to accelerate I/O in distributed DL training workflows.

d) ***Comparison with Existing Systems:*** Existing HPC caching frameworks [29], [30], [34], [49], [57]–[61] address parts of the I/O bottleneck in distributed deep learning. However, none of them provide portability, transparency, and

efficient multi-tier data retrieval at scale. For example, HVAC uses a fixed NVMe cache and cannot adapt to heterogeneous memory hierarchies. Other systems depend on custom APIs, modified file systems, or centralized metadata services. These design choices limit deployment on production HPC platforms.

In contrast, FITCACHE coordinates a dynamic multi-tier cache hierarchy and issues concurrent fetches across tiers. It selects the fastest response at runtime. This design tolerates device-level performance variability and does not rely on centralized metadata or static access paths. Moreover, FITCACHE preserves POSIX semantics and requires no changes to existing training code. This approach enables transparent deployment on large-scale, heterogeneous systems. These differences distinguish FITCACHE from prior HPC caching frameworks and motivate its design.

The comparison of FITCACHE and existing cache systems for distributed DL is detailed in Table I. In this table, *Tiered Systems* indicate support for coordinating multiple memory and storage layers (e.g., DRAM, PMem, and NVMe) within one framework, and *Parallel Fetch* indicates issuing concurrent reads across tiers and selecting the fastest response.

III. FITCACHE: DESIGN AND IMPLEMENTATION

A. Problem Formulation

1) Basic Setup: Let the training dataset be

$$D = \{s_1, \ldots, s_M\},$$

with each sample accessed once per epoch, where M is the total number of samples. Let $\mathcal{T} = \{t_1, \ldots, t_L\}$ denote the set of L available cache tiers (*e.g.*, memory or NVMe), and let PFS denote the underlying parallel file system. We define the full storage hierarchy as $\mathcal{T} \cup \{\mathrm{PFS}\}$.

For any sample $s \in D$ and storage tier $t \in \mathcal{T}$, let $\tau_t(s) > 0$ denote the empirically measured end-to-end latency to retrieve s from t during training.

Let $T_{\mathrm{comp}} > 0$ denote the compute time per epoch (*i.e.*, forward, backward, optimizer, and collectives), and let $E \in \mathbb{N}$ be the total number of training epochs.

2) Best-case I/O Latency per Sample: We define the cache-tier-limited access time for each sample as

$$\tau^*(s) = \min_{t \in \mathcal{T}} \tau_t(s),$$

which represents the fastest empirically observed time to retrieve s from any available storage tier.

Since each sample is read once per epoch, the per-epoch I/O lower bound is

$$J^*_{\mathrm{io}} = \sum_{s \in D} \tau^*(s),$$

capturing the minimum I/O time needed to load all samples during one epoch if each sample is read from its fastest available tier.

3) Epoch and Total Training Time: Given a fixed compute time per epoch T_{comp}, the achievable epoch time and total training time satisfy

$$T_{\mathrm{epoch}} \;\geq\; \max\{\, T_{\mathrm{comp}},\; J^*_{\mathrm{io}} \,\}, T_{\mathrm{total}} \;\geq\; E{\cdot}\max\{\, T_{\mathrm{comp}},\; J^*_{\mathrm{io}} \,\}.$$

$$\text{subject to} \quad s \text{ is retrieved from } \arg\min_{t \in \mathcal{T}} \tau_t(s)$$

4) Optimization Objectives: Our goal is to minimize both the per-epoch I/O cost and the total training time

$$\min\; J^*_{\mathrm{io}}, \qquad \min\; T_{\mathrm{total}}.$$

Any reduction in J^*_{io} directly tightens the lower bound on T_{epoch} and T_{total}, aligning with our goal of improving end-to-end training performance.

B. Design Overview

The problem formulation above shows that the per-epoch training time is lower-bounded by the larger of the compute cost and the I/O cost. Reducing J^*_{io} is therefore essential to improving end-to-end performance. To achieve this, we design FITCACHE, a portable and drop-in multi-tier caching framework that exploits existing storage resources *without any* modification to DL applications and underlying PFS.

In typical HPC settings, FITCACHE exposes three storage tiers: a high-latency parallel file system (PFS), a low-latency node-local storage layer, and an ultra-low-latency memory layer. While FITCACHE is not limited to just these three tiers, we adopt this structure for clarity throughout the paper. As shown in Figure 3, the lower-latency tiers offer smaller capacity. This tier asymmetry introduces fundamental challenges in data placement and caching pressure, especially for large training workloads that reuse the same data across multiple epochs. Note that, to address heterogeneous hardware environments, FITCACHE dynamically adapts its hierarchy. For example, when a system lacks node-local NVMe, it seamlessly leverages CPU memory and compute resources to cooperatively cache and serve data, ensuring consistent acceleration and sustained caching efficiency across diverse HPC configurations.

To further reduce read latency across tiers, FITCACHE introduces a multi-tier interposing mechanism. When a file is requested during training, the runtime issues parallel read operations to all candidate storage tiers, such as memory and NVMe, without assuming prior knowledge of the file's location. The system returns the data from the fastest responding tier and ignores the remaining responses. By leveraging multiple tiers concurrently, this design helps to improve the response time and reduce the data movement overhead. The multi-tier interposing mechanism is particularly effective in HPC environments where resource contention or background I/O noise can introduce latency variability while remaining transparent to the application.

Our system consists of three key components:

Client-side Interception. To achieve application transparency, each compute node deploys a lightweight interception client that dynamically redirects I/O operations through

LD_PRELOAD. This client intercepts system calls such as open, read, and close at runtime, forwarding these operations to the FITCACHE server over a high-speed RPC channel. All other operations, including metadata, mmap, and directory calls, are directed at the native filesystem without interception. The client also checks open flags to avoid writable or unsafe accesses. By operating at the syscall level, FITCACHE remains agnostic to the underlying DL framework (*e.g.*, TensorFlow, PyTorch) and seamlessly integrates into existing workflows. This interception mechanism enables FITCACHE's full compatibility with distributed DL training pipelines.

Hierarchical Cache Tier. Each server manages a multi-level cache composed of PMem (as fsdax), memory (as tmpfs), NVMe, and PFS (*e.g.*, BeeGFS [62] and Lustre [63]). On first access, data is read from PFS and placed into fast local tiers. A capacity-aware data placement policy determines if data is stored in any specific file caching tier based on the access times and storage capacity. Future reads are served from this file cache, ensuring the lowest possible access latency.

Interpose Multi-Tier Fetch. Beyond simple caching, FITCACHE implements a multi-tier interposing protocol to accelerate data delivery. When a requested block may reside in multiple tiers or storage nodes, the FITCACHE server issues concurrent fetches to each candidate location and returns the earliest response to the client. The protocol also ignores all other requests once the first reply arrives, preventing unnecessary bandwidth consumption. This mechanism ensures that DL jobs always receive data from the fastest available source, maintaining high GPU utilization even under fluctuating load across thousands of nodes.

C. Memory-Tier Abstraction

Modern HPC systems increasingly integrate multiple storage tiers with distinct trade-offs in latency, bandwidth, and capacity. At the top of this hierarchy lies the memory tier. For example, DRAM and PMem. DRAM provides nanosecond-scale access latency and high bandwidth, making it ideal for caching frequently used training data. In contrast, PMem offers higher capacity with slightly increased latency, but still significantly faster access than disk or SSD-based storage.

To accommodate diverse hardware configurations, FITCACHE provides a flexible abstraction for the memory tier that supports a wide range of fast-access storage media commonly found in HPC systems. These include volatile memory technologies like DRAM and non-volatile options such as PMem, among others. Our system is not limited to any specific memory technology and adapts to available resources at runtime. In our evaluation (see SectionIV), we demonstrate this flexibility by deploying FITCACHE on systems with DRAM-based memory tiers and systems with PMem-backed tiers. To use the memory tier as a high-speed cache, FITCACHE stages data in temporary memory-backed file systems, depending on system support. When a training sample is accessed for the first time, it is cached to the fastest available memory tier. The caching logic relies on per-tier capacity thresholds to control space usage and maintain performance stability.

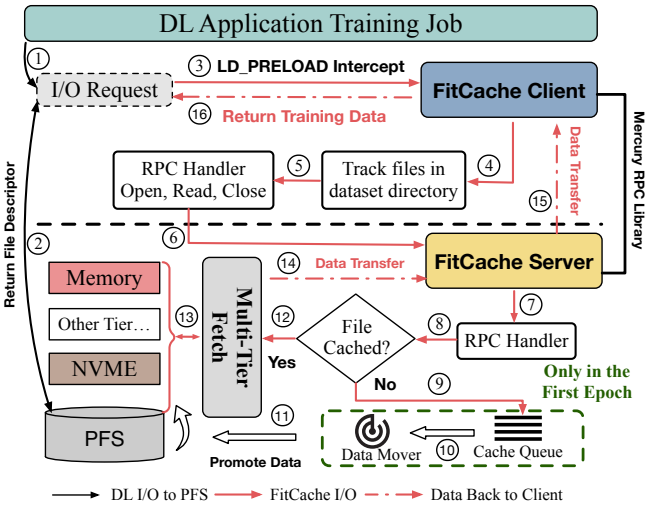


Fig. 4: FITCACHE I/O flow on compute node. Note that the DL applications and underlying PFS are unmodified. FITCACHE client and server instances can be co-located on the same node or on different nodes based on the underlying target architecture. Further, multiple FITCACHE server instances can be executed on a single node.

As a prior study [64] has shown that SSD and NVMe performance significantly degrades as device utilization approaches full capacity, FITCACHE limits usage of lower tiers such as NVMe to a fixed fraction of their total capacity (*e.g.*, 60–80%). FITCACHE conservatively restricts usage to below saturation levels, thereby maintaining stable I/O performance. This approach allows the system to exploit memory-tier speed for frequently accessed data while ensuring predictable performance. As a result, FITCACHE delivers consistent caching behavior across a range of hardware platforms without requiring application modifications or device-specific tuning.

D. Multi-Tier Interposing for Data Retrieval

1) Multi-Tier Fetch: To accelerate data delivery across heterogeneous storage tiers, FITCACHE introduces a multi-tier fetch mechanism that issues concurrent read requests to all possible file locations within a single server. The target server is selected using a consistent hash of the file path. These candidate tiers include memory-backed file systems, node-local NVMe, and the PFS. Once the first response is received, the system immediately returns the data to the client and ignores other outstanding requests, ensuring minimal latency without stalling on slower devices.

FITCACHE operates as a transparent, drop-in caching layer between DL applications and the underlying storage infrastructure. Common DL frameworks typically interact with storage through standard POSIX interfaces (open, read, close) on read-only datasets stored in shared PFS. By interposing these system calls via a preloadable user-space library (LD_PRELOAD), FITCACHE redirects I/O to its caching runtime without requiring changes to the training workflows.

The multi-tier fetch protocol only relies on the runtime latency to guide tier selection. This design allows FITCACHE to automatically bypass stragglers and exploit faster storage paths as they become available, without maintaining complex metadata or making assumptions about data residency. Even under load imbalance or performance variability, FITCACHE ensures that each read request results in a single data delivery, avoiding redundant transfers and minimizing contention.

2) Data Flow: Figure 4 depicts the end-to-end data path in each case. Without FITCACHE, DL training jobs issue file system calls directly to the PFS, as shown in the baseline path (❶ → ❷). This direct access path incurs high latency and exposes jobs to I/O contention, particularly at scale. In contrast, we illustrate the I/O execution flow under FITCACHE using two representative scenarios: the initial access during the first epoch and subsequent accesses in later epochs.

First Epoch Data Flow: After the DL application initiates a read request to the DL dataset directory on PFS, the FITCACHE client intercepts any incoming file I/O <open, read and close>. Once intercepted, the client identifies the dataset directory and begins tracking the accessed files (❶, ❸, and ❹). The RPC handler of the FITCACHE client redirects the requested file I/O to the appropriate FITCACHE server, either local or remote. This redirection is guided by a hash of the file path to select the target server. Client and server each manage their own RPC handlers, which send and receive messages over the network (❺ → ❼). The server invokes its RPC handler to check whether the requested file is already cached (❽). Since this is the first access, the file is not yet available in the cache. The server inserts the file path into a shared queue, which is monitored by a background data mover thread. The thread fetches the file from the PFS and promotes it into the node-local multi-tier storage system (❾ → ⓫). The file content is returned to the client (⓮), which passes the data to the DL application as if it were served directly by the file system (⓯ → ⓰).

Subsequent Epoch Data Flow: FITCACHE reuses the same interception and request forwarding steps. However, once the server detects that the requested file is already cached, it bypasses the PFS entirely (❾ → ⓭). The data is retrieved from the fastest tier and returned to the client and application with minimal delay (⓭ → ⓯). This design ensures that each file is fetched from the PFS at most once, while later accesses benefit from local caching and reduced latency.

E. FITCACHE *Servers selection*

Given the caching and multi-tier fetch design discussed above, a natural question arises: How many FITCACHE servers should we launch to ensure fast data delivery without excessive resource usage? This decision is critical to achieving our optimization goals in Section III-A, particularly minimizing total training time and ensuring that I/O does not become a bottleneck. In this subsection, we present a simple sizing rule to determine the minimum number of servers needed, based on empirical throughput and per-epoch data volume.

Problem Notation. Let $w(s) > 0$ denote the size of sample s. Then the per-epoch data volume is

$$V_{\text{epoch}} \triangleq \sum_{s \in D} w(s).$$

Let $\widehat{B}_{\text{srv}}$ be the *measured* sustained end-to-end throughput (bytes/s) of a single FITCACHE server on the target system, evaluated along the same runtime path as training (including RPC and multi-tier fetch). Let $\eta \in (0, 1]$ be a safety factor (*e.g.*, $\eta = 0.7$–0.85) to absorb variability and interference. Let T_{comp} denote the per-epoch compute time (as in Section III-A). We set an I/O time budget

$$T_{\text{budget}} \triangleq \beta T_{\text{comp}}, \qquad \beta > 0,$$

where $\beta = 1$ enforces "I/O no slower than compute" and smaller β is more aggressive. Let $S \in \mathbb{N}$ be the number of FITCACHE servers to launch.

Sizing rule. With S servers, the sustained aggregate throughput is at least $S\,\eta\,\widehat{B}_{\text{srv}}$. Hence the time to move one-epoch data is upper bounded by

$$\frac{V_{\text{epoch}}}{S\,\eta\,\widehat{B}_{\text{srv}}}.$$

To ensure that the per-epoch I/O time does not exceed the budget used in Section III-A, choose

$$S = \left\lceil \frac{V_{\text{epoch}}}{\eta\,\widehat{B}_{\text{srv}}\,T_{\text{budget}}} \right\rceil.$$

Example. Suppose the dataset is $2\,\text{TB}$ in size and we measured the sustained throughput of a single server as $\widehat{B}_{\text{srv}} = 15\,\text{GB/s}$ (including RPC and multi-tier fetch). We set a safety factor $\eta = 0.75$ and a budget ratio $\beta = 1.0$ for $T_{\text{budget}} = 100\,\text{s}$, assuming $T_{\text{comp}} = 100\,\text{s}$. Then the number of servers required is

$$S = \left\lceil \frac{2 \times 10^{12}}{0.75 \times 15 \times 10^{9} \times 100} \right\rceil = \lceil 1.78 \rceil = 2.$$

In this configuration, launching two servers per compute node is sufficient to keep per-epoch I/O time under budget. Based on our measurements, running 2 to 4 FITCACHE servers per node provides enough performance (detailed in Section IV-F).

F. Data Residency and Coordination across Tiers

In a dynamic runtime system like FITCACHE, it is essential to determine how and where data is placed across the storage tiers during execution. FITCACHE adopts a simple but effective strategy based on a FIFO (First-In, First-Out) placement policy, with tier-specific capacity thresholds to prevent performance degradation at high utilization.

On first access, each file is fetched from PFS and cached into the fastest available local tier (*i.e.*, memory or NVMe) based on current space availability. As training progresses, reads are served from the tier offering the lowest access latency, falling back to lower tiers only when necessary. To ensure performance stability, FITCACHE limits NVMe usage to a fixed fraction of its capacity (*e.g.*, 80%).

Two-Level Hash Bin. In addition to placement policies, managing the physical organization of cached files is critical at scale. Deep learning datasets may consist of millions of files, and storing all cached files in a flat directory structure (as done in prior systems such as HVAC [30]) can cause severe inode lookup overhead and degrade file system performance. To address this, FITCACHE adopts a two-level hash binning strategy that uniformly distributes cached files across nested subdirectories.

Specifically, for each original file path, FITCACHE computes a 64-bit hash and uses the lowest two bytes to define a two-level directory hierarchy. The upper and lower 8 bits of the hash determine two subdirectory names in hexadecimal, resulting in a file layout of the form "base/s1/s2", *i.e.*, a base path followed by two nested subdirectories, s1 and s2. This scheme ensures uniform file distribution, reduces inode contention, and improves I/O parallelism in large-scale training workloads.

IV. EVALUATION

A. Experimental Setup

1) Testbed: We evaluate FITCACHE on two platforms: the Frontier supercomputer and a smaller institutional facility referred to as GPUCLUSTER in the following. Frontier is based on the HPE Cray EX architecture and is deployed at the Oak Ridge Leadership Computing Facility (OLCF). It consists of 9,402 compute nodes. Each node has one AMD EPYC 8124P CPU with 64 cores and four AMD MI250X GPUs, each with two graphics compute dies (GCDs) accessible as eight GCDs. The job scheduler refers to the GCDs as GPUs, as do we in the following. Each node also provides 512 GB of system memory and two local NVMe drives (1.92TB each), which form a high-throughput node-local storage tier. All nodes are connected to Orion, a 700PB Lustre parallel file system. This configuration allows us to evaluate our caching framework on a production-scale exascale platform with real DL workloads.

The smaller GPUCLUSTER contains 80 compute nodes, each with different CPU and GPU configurations. For our experiments, each selected node includes one RTX 4060 Ti 16GB GPU and one AMD EPYC 8124P CPU with 16 cores. The cluster also features two PMem nodes, each with 512GB of phase change memory in DRAM slots. We exploit this PMem as a storage tier in our evaluation.

2) Application and Datasets: We selected a mix of deep neural networks and scientific deep learning applications from the MLPerf-HPC benchmark suite [65] to evaluate the scalability of our caching framework.

- CosmoFlow [43] is a 3D convolutional neural network from MLPerf HPC v0.5. It predicts physical parameters of the universe from N-body cosmology simulation data. We train CosmoFlow using the cosmoUniverse dataset, which includes over 51,000 model parameters.
- DeepCAM [66] is a PyTorch-based model for climate segmentation. It is part of the Exascale Deep Learning for Climate Analytics project. DeepCAM is trained on images of size 768x1152 with 16 channels. This input size is much larger than standard vision datasets like ImageNet, where images average 469x387 pixels and have at most four channels.

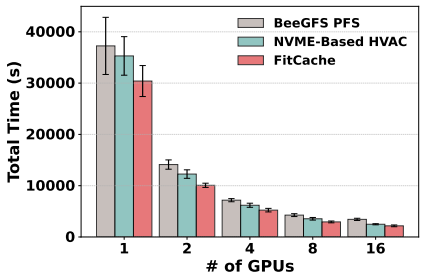


Fig. 5: Small-scale experiments on GPUCLUSTER

3) Datasets: We use two large-scale datasets in our experiments. The first is cosmoUniverse, which contains preprocessed TFRecord files generated from simulation runs by the ExaLearn group at NERSC [43]. It includes 524,288 training samples and 65,536 validation samples, with a total size of 1.3 TB. The second dataset comes from CAM5 [67] climate simulations, also hosted at NERSC. The samples are stored in HDF5 format, with a total size of more than 2.4 TB. Each input image has a shape of (768,1152,16), and each label has a shape of (768,1152) with three classes: background, atmospheric river, and tropical cyclone. The labels are generated using TECA [68]. Both datasets are large enough to create significant I/O pressure in scale-out settings, and they expose limitations of shared file systems such as GPFS. We repeat all experiments three times unless stated otherwise and report the average with a 95% confidence interval.

For the evaluation, we compare the following systems:

Lustre Orion PFS: This is the default PFS on the Frontier supercomputer, where all training datasets are read directly from the Lustre Orion PFS.

BeeGFS PFS: This is the default PFS on GPUCLUSTER, where all training datasets are read directly from BeeGFS.

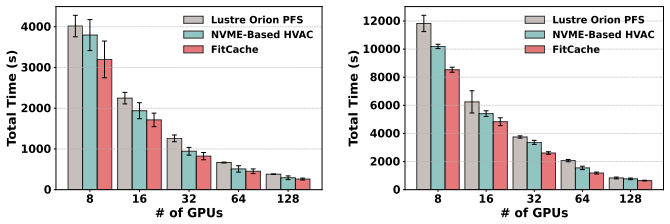
NVME-based HVAC [30]: The caching system HVAC implementation that stages data exclusively on local NVMe drives, without using system memory.

FITCACHE: Our proposed multi-tiered caching system. We use `tmpfs` as the memory cache tier on Frontier and PMem as the memory tier on the small-scale HPC system.

B. Small-Scale Experiments on GPUCLUSTER

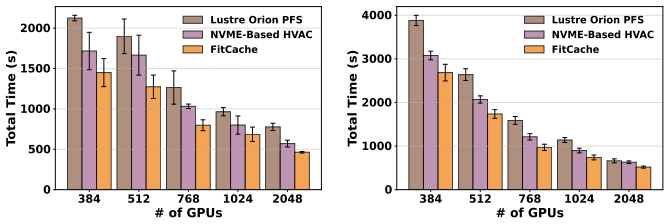
To evaluate the impact of FITCACHE under a small cluster and demonstrate its functionality, we conduct small-scale experiments on GPUCLUSTER using 1-16 GPUs and compare training times across three configurations: default BeeGFS, NVMe-backed HVAC, and our FITCACHE. Results are summarized in Figure 5.

Across all scales, FITCACHE consistently achieves lower training times. For instance, on a single GPU, FITCACHE reduces training time from 37,254.62s (BeeGFS) to 30,401.83s, achieving an 18.4% speedup. While all systems benefit from



(a) CosmoFlow (b) DeepCAM

Fig. 6: Small-scale experiment results on Frontier



(a) CosmoFlow (b) DeepCAM

Fig. 7: Large-scale experiment results on Frontier

increasing GPU count, FITCACHE maintains its advantage. With 16 GPUs, training time is reduced to 2,172.51s with FITCACHE, compared to 3436.67s (BeeGFS) and 2481.28s (HVAC), resulting in speedups of 36.8% and 12.4%, respectively.

The improvement margins are also prominent at small scales (1-4 GPUs). Although only two PMem nodes are available, each GPU is assigned two FITCACHE servers. This mapping ensures sufficient caching bandwidth and avoids contention, particularly at small scales. For example, FITCACHE achieves a 28.6% speedup over BeeGFS and a 17.9% speedup over HVAC with 2 GPUs. This shows that even under smaller GPU counts, where performance variability is typically higher, FITCACHE consistently delivers strong speedups.

These results confirm that even under modest parallelism, a hybrid memory-tier cache brings measurable benefits. FITCACHE delivers both lower latency and improved throughput, enabling faster iteration and reduced job turnaround time on smaller shared clusters like GPUCLUSTER.

C. Small-Scale Experiments on Frontier

We evaluate the scalability and effectiveness of FITCACHE on Frontier for the small-scale tests, scaling from 8-128 GPUs. Each node runs 2 FITCACHE servers and leverages memory-based caching. Figure 6 presents the results alongside two baselines: the Lustre Orion PFS (no caching) and the original HVAC design using NVMe-based caching.

For CosmoFlow, as shown in Figure 6a, FITCACHE consistently achieves lower training time across all scales. At small scale (8 GPUs), FITCACHE already outperforms both baselines, reducing training time by 20.4% compared to PFS. As the scale increases, the benefit becomes more pronounced. At 16 GPUs, FITCACHE speeds up training by 23.8% over PFS, cutting time from 2247.39s to 1713.53s. And at 128 GPUs, it brings the time down to 260.57s, compared to 292.90s with HVAC and 380.47s with PFS.

For DeepCAM, as shown in Figure 6b, the benefit of memory-tier caching is even more prominent. At 32 GPUs, FITCACHE achieves a 30.4% reduction over PFS and 22.5% over HVAC. At 64 GPUs, the gain is even achieved at a notable 42.7% when compared to the Lustre Orion PFS. While all methods benefit from increased parallelism, the relative improvement from memory caching grows with scale, reflecting its ability to mitigate backend storage contention and reduce I/O wait time even as the number of concurrent read requests rises sharply.

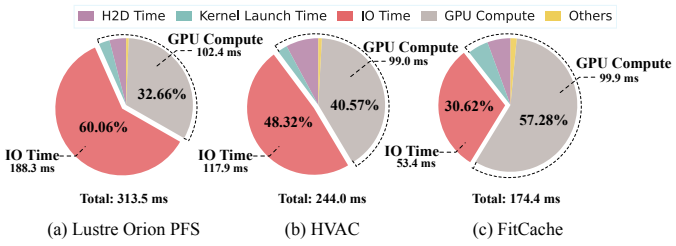
Overall, FITCACHE's DRAM+NVMe caching delivers an average improvement of 29.2% for DeepCAM and 28.5% for CosmoFlow over Lustre Orion PFS runs, with further gains over NVMe-only HVAC. These results confirm that memory-tier caching is particularly effective at scale, where even partial caching of frequently accessed samples can reduce read latency, increase overlap between I/O and compute, and improve GPU utilization. By transparently staging data in the fastest available tier and avoiding redundant I/O, FITCACHE offers a practical and portable performance boost across a range of model sizes and access patterns.

D. Large-Scale Experiments on Frontier

To evaluate the effectiveness of FITCACHE at scale, we conduct large-scale training experiments across up to 2048 GPUs on Frontier. Each configuration leverages two FITCACHE servers per node, and all methods are compared under identical conditions. Results are shown in Figure 7.

As GPU count increases, the pressure on backend storage subsystems (*e.g.*, Lustre) becomes more pronounced. In such settings, the ability to absorb high I/O concurrency becomes critical. FITCACHE consistently demonstrates superior scalability over both baselines. For CosmoFlow, while the Lustre Orion PFS beyond 1024 GPUs improves by only ≈19% when doubling from 1024 to 2048 GPUs. However, FITCACHE continues to scale, reducing training time by over 32% in the same step (from 685.05s to 463.72s). This trend highlights that backend I/O contention becomes the limiting factor at scale for traditional approaches, while FITCACHE sidesteps this bottleneck via memory-tier caching and concurrent multi-tier fetch.

The trend is even more prominent for DeepCAM, which exhibits heavier I/O behavior. At 2048 GPUs, FITCACHE completes an epoch in 518.02 seconds, compared to 661.14 seconds for the Lustre Orion PFS and 629.35 seconds for NVMe-backed HVAC. Notably, the absolute gap widens as scale increases: At 384 GPUs, FITCACHE improves over HVAC by 12.8%. This gap expands to 17.7% at 2048 GPUs—despite higher concurrency and heavier load. These results suggest that the NVMe tier is not enough at scale, while memory tiers in FITCACHE absorb peak traffic more effectively. Rather than flattening or regressing under large-scale pressure, FITCACHE



(a) Lustre Orion PFS (b) HVAC (c) FitCache

Fig. 8: Per-batch IO reduction on Frontier

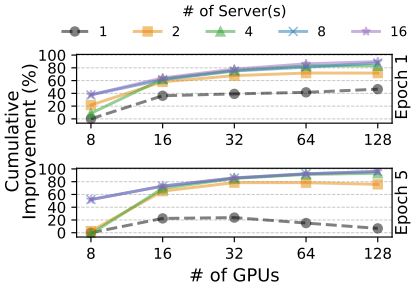


Fig. 9: % Improvement Rate on different server numbers

continues to provide near-linear improvements, demonstrating its robustness under extreme load. This behavior is attributed to our core design choice of the multi-tier fetch approach, which masks tail latency from any single tier.

In summary, FITCACHE's advantages become more pronounced at scale, where traditional storage and caching methods struggle with throughput saturation and file system contention. The system's design enables it to convert additional GPUs into effective throughput gains. This results in up to 40% speedup over Lustre Orion PFS runs and outperforms NVMe-backed HVAC by a significant margin. Crucially, the results reveal improved system throughput and better GPU utilization at scale. By removing I/O bottlenecks, FITCACHE allows DL jobs to fully utilize additional compute resources. This enables faster time-to-solution for large-scale scientific workloads, reduces queuing overhead on shared systems.

E. Per-batch Time Breakdown

To evaluate the runtime impact of memory-tier caching at scale, we profiled per-batch training time (batch size = 2) on 1024 GPUs into five components: host-to-device (H2D) transfer, kernel launch, I/O, GPU compute, and miscellaneous overheads, as shown in Figure 8. All configurations used two FITCACHE servers per node.

In the Lustre Orion PFS, I/O dominates the batch time at 188.3 ms, accounting for over 60% of total batch time. NVMe-based HVAC reduces this to 117.9 ms, and FITCACHE further cuts it down to 53.4 ms, a 71.6% improvement compared to Lustre and 54.7% over HVAC. These results confirm the effectiveness of memory-based caching in alleviating backend storage bottlenecks, especially under high concurrency.

Kernel launch time is also affected by I/O latency. HVAC slightly reduces it to 5.8 ms, while FITCACHE achieves 8.7 ms, on par with Lustre. H2D transfer sees moderate variations, with FITCACHE at 9.8 ms, which is lower than Lustre (12.6 ms) and significantly lower than HVAC (19.2 ms). GPU compute time remains stable across all setups, suggesting caching does not interfere with accelerator-side execution.

Overall, FITCACHE enables faster and more consistent data delivery at scale, shortening I/O stalls and reducing variability in host-side execution. These improvements translate into tighter compute scheduling, less idle GPU time, and improved training throughput on leadership-scale systems.

F. Node-to-Server Ratio

To quantify the effect of FITCACHE server instances on end-to-end performance, we conduct an ablation study where the number of servers per compute nodes varies from 1 to 16. We report cumulative training time improvement over Lustre Orion PFS for two representative stages: the first epoch (which incurs only cache misses) and the fifth epoch (where the cache is warm). Results are summarized in Figure 9.

During the first epoch, increasing the number of servers improves performance. With 1 server, gains are marginal across all scales. However, moving to 2 servers already provides 58.06%-71.82% improvement depending on compute scale. Further scaling to 16 servers yields up to 89.79% improvement over the Lustre Orion PFS at 128 GPUs. The results indicate that more servers help absorb the initial I/O burst more effectively by parallelizing fetches from PFS and accelerating the initial population of the cache.

By the fifth epoch, the system transitions into a regime where cache hits dominate. Here, the performance stabilizes with diminishing returns beyond 4-8 servers. For instance, at 4 nodes, improvement increases from 69.56% (2 servers) to 85.21% (4 servers), and further to 93.22% with 16 servers. However, the incremental benefit reduces after 8 servers. This suggests that once the cache is sufficiently populated, I/O becomes less of a bottleneck and the need for aggressive fetch parallelism diminishes.

In summary, server-side parallelism plays a crucial role in early-stage cache population (Epoch 1), where fetch bandwidth limits performance. However, after warming up the cache (Epoch 5), even moderate server counts (2–4 per node) are sufficient to sustain high performance, consistent with our analytical sizing rule (see Section III-E). This highlights a favorable cost-benefit tradeoff in practical deployments.

G. Cross-Tier Performance Discussion and Extrapolation

We quantify the impact of different storage tiers on end-to-end training speed by comparing latency (Lat, in microseconds) and bandwidth (BW, in GB/s) characteristics across platforms. Tables II and III summarize empirical measurements on GPUCLUSTER (BeeGFS) and Frontier (Lustre), respectively.

On Frontier, where training is I/O-intensive and dominated by reads from Lustre, using node-local DRAM (via `tmpfs`) yields a 1.39× average speedup in CosmoFlow training across

Compute	Lat (μs)	BW (GB/s)	Lat Accel.	BW Accel.	Train Speedup
PMem [36]	0.305	39.4	983.6×	2.81×	1.24×
NVMe [69]	~ 80	~ 6	3.75×	2×	1.14×
BeeGFS [70]	~ 300	~ 3	1×	1×	1×

TABLE II: Acceleration Rate on GPUCLUSTER (BeeGFS as baseline)

Compute	Lat (μs)	BW (GB/s)	Lat Accel.	BW Accel.	Train Speedup
DRAM [36]	0.081	120	6170×	40×	1.39×
NVMe [69]	~ 80	~ 6	6.25×	2×	1.27×
Lustre [71]	~ 500	~ 3	1×	1×	1×

TABLE III: Acceleration Rate on Frontier (Lustre as baseline)

large-scale runs (384-2048 GPUs). This corresponds to a $6170\times$ latency reduction and $40\times$ bandwidth increase over Lustre, suggesting that memory latency and concurrency are key bottlenecks at scale. In comparison, NVMe delivers more modest gains, namely $1.27\times$ speedup, $6.25\times$ latency reduction, and $2\times$ bandwidth improvement.

While DRAM offers the best performance, it is volatile and typically limited in capacity. PMem offers a middle ground as it provides orders-of-magnitude lower latency than NVMe and higher capacity than DRAM. Although PMem is not available on Frontier, we can extrapolate from GPUCLUSTER results: On GPUCLUSTER, PMem achieves a $983.6\times$ latency gain and $2.81\times$ bandwidth gain over BeeGFS, leading to a $1.24\times$ training speedup.

Since Frontier does not support PMem, we estimate its potential benefit based on observed workload characteristics and performance trends. Given a similar access patterns, we project that incorporating PMem into Frontier would yield a training speedup between NVMe and DRAM. Specifically, we estimate an average end-to-end speedup of 1.30-1.35×, exceeding NVMe performance while approaching that of DRAM. This extrapolation suggests that hybrid memory-storage architectures combining PMem and NVMe could further enhance training throughput on leadership-scale systems, particularly when DRAM capacity is insufficient to cache the full dataset. Alternatively, a three-tiered approach could combine DRAM, PMem and NVMe into one caching abstraction boasting overall capacity to an even higher level.

V. CONCLUSION

In this paper, we presented FITCACHE, a transparent drop-in caching framework designed to accelerate distributed deep learning (DL) workloads. FITCACHE orchestrates heterogeneous memory and storage tiers—including DRAM, Persistent Memory (PMem), and NVMe into a unified multi-tier cache above the parallel file system. By issuing concurrent I/O requests across tiers and applying adaptive cache placement, FITCACHE effectively minimizes latency, tolerates hardware variability, and delivers substantial speedups in large-scale training on HPC systems. Our design includes a lightweight client-side interception mechanism and an adaptive data placement with capacity-aware policies. Evaluations across both large-scale (Frontier) and small-scale HPC clusters demonstrate significant performance gains, cutting training time by up to 40% and reducing per-batch I/O latency by up to 71.6%. Overall, FITCACHE addresses key I/O challenges in distributed DL training, offering a drop-in acceleration layer that adapts to diverse system architectures and workloads.

ACKNOWLEDGMENT

This work was funded in part by subcontract B664613 from LLNL, as well as NSF grants CISE-2217020 and CISE-2316201. This research also used resources of the Oak Ridge Leadership Computing Facility, located at the National Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the DOE under Contract DE-AC05-00OR22725.

REFERENCES

[1] A. Cruz-Roa, H. Gilmore, A. Basavanhally, M. Feldman, S. Ganesan, N. N. Shih, J. Tomaszewski, F. A. González, and A. Madabhushi, "Accurate and reproducible invasive breast cancer detection in whole-slide images: A deep learning approach for quantifying tumor extent," *Scientific reports*, vol. 7, no. 1, p. 46450, 2017.

[2] O. L. Saldanha, J. Zhu, G. Müller-Franzes, Z. I. Carrero, N. R. Payne, L. Escudero Sánchez, P. C. Varoutas, S. Kyathanahally, N. G. Laleh, K. Pfeiffer *et al.*, "Swarm learning with weak supervision enables automatic breast cancer detection in magnetic resonance imaging," *Communications medicine*, vol. 5, no. 1, p. 38, 2025.

[3] M. Rajesh, R. G. Babu, U. Moorthy, and S. V. Easwaramoorthy, "Machine learningdriven framework for realtime air quality assessment and predictive environmental health risk mapping," *Scientific Reports*, vol. 15, no. 1, p. 28801, 2025.

[4] Z. Lv, D. Chen, and H. Lv, "Smart city construction and management by digital twins and bim big data in covid-19 scenario," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 18, no. 2s, pp. 1–21, 2022.

[5] S. Wu, "Spatiotemporal dynamic forecasting and analysis of regional traffic flow in urban road networks using deep learning convolutional neural network," *IEEE transactions on intelligent transportation systems*, vol. 23, no. 2, pp. 1607–1615, 2021.

[6] A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan *et al.*, "Deepseek-v3 technical report," *arXiv preprint arXiv:2412.19437*, 2024.

[7] A. Koubaa, W. Boulila, L. Ghouti, A. Alzahem, and S. Latif, "Exploring chatgpt capabilities and limitations: A critical review of the nlp game changer," 2023.

[8] M. Chai, E. Herron, E. Cervantes, and T. Ghosal, "Exploring scientific hypothesis generation with mamba," in *Proceedings of the 1st Workshop on NLP for Science (NLP4Science)*, L. Peled-Cohen, N. Calderon, S. Lissak, and R. Reichart, Eds. Miami, FL, USA: Association for Computational Linguistics, Nov. 2024, pp. 197–207. [Online]. Available: https://aclanthology.org/2024.nlp4science-1.17/

[9] W. Gao, X. Liu, F. Wang, D. Lu, and J. Yin, "Decoding memories: An efficient pipeline for self-consistency hallucination detection," 2025. [Online]. Available: https://arxiv.org/abs/2508.21228

[10] M. Goldblum, H. Souri, R. Ni, M. Shu, V. Prabhu, G. Somepalli, P. Chattopadhyay, M. Ibrahim, A. Bardes, J. Hoffman *et al.*, "Battle of the backbones: A large-scale comparison of pretrained models across computer vision tasks," *Advances in Neural Information Processing Systems*, vol. 36, pp. 29 343–29 371, 2023.

[11] X. Zhao, L. Wang, Y. Zhang, X. Han, M. Deveci, and M. Parmar, "A review of convolutional neural networks in computer vision," *Artificial Intelligence Review*, vol. 57, no. 4, p. 99, 2024.

[12] MarketsandMarkets. (2024) Conversational ai market worth $49.9 billion by 2030. [Online]. Available: https://www.prnewswire.com/news-releases/conversational-ai-market-worth-49-9-billion-by-2030---exclusive-report-by-marketsandmarkets-302125862.html

[13] V. Farhadi, F. Mehmeti, T. He, T. F. La Porta, H. Khamfroush, S. Wang, K. S. Chan, and K. Poularakis, "Service placement and request scheduling for data-intensive applications in edge clouds," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 779–792, 2021.

[14] H. B. Abdalla, "A brief survey on big data: technologies, terminologies and data-intensive applications," *Journal of Big Data*, vol. 9, no. 1, p. 107, 2022.

[15] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 873–880.

[16] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," *Advances in neural information processing systems*, vol. 25, 2012.

[17] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[18] Oak Ridge Leadership Computing Facility, "Frontier Supercomputer," https://www.olcf.ornl.gov/frontier/, 2025.

[19] X. Ding, L. Chen, M. Emani, C. Liao, P.-H. Lin, T. Vanderbruggen, Z. Xie, A. Cerpa, and W. Du, "Hpc-gpt: Integrating large language model for high-performance computing," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 951–960.

[20] Y. Fan, Z. Lan, P. Rich, W. Allcock, and M. E. Papka, "Hybrid workload scheduling on hpc systems," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 470–480.

[21] X. Cao, T. Başar, S. Diggavi, Y. C. Eldar, K. B. Letaief, H. V. Poor, and J. Zhang, "Communication-efficient distributed learning: An overview," *IEEE journal on selected areas in communications*, vol. 41, no. 4, pp. 851–873, 2023.

[22] K. Z. Ibrahim, T. Nguyen, H. A. Nam, W. Bhimji, S. Farrell, L. Oliker, M. Rowan, N. J. Wright, and S. Williams, "Architectural requirements for deep learning workloads in hpc environments," in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2021, pp. 7–17.

[23] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and mitigating data stalls in dnn training," *Proc. VLDB Endow.*, vol. 14, no. 5, p. 771–784, Jan. 2021. [Online]. Available: https://doi.org/10.14778/3446095.3446100

[24] S. Pumma, M. Si, W.-C. Feng, and P. Balaji, "Scalable deep learning via i/o analysis and optimization," *ACM Trans. Parallel Comput.*, vol. 6, no. 2, 2019. [Online]. Available: https://doi.org/10.1145/3331526

[25] P. Braam, "The lustre storage architecture," 2019. [Online]. Available: https://arxiv.org/abs/1903.01955

[26] R. Macedo, C. Correia, M. Dantas, C. Brito, W. Xu, Y. Tanimura, J. Haga, and J. Paulo, "The case for storage optimization decoupling in deep learning frameworks," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 649–656.

[27] M. Hoseinzadeh, "A survey on tiering and caching in high-performance storage systems," 2019. [Online]. Available: https://arxiv.org/abs/1904.11560

[28] A. Khan, C. Zimmer, S. Atchley, R. Miller, S. Oral, and F. Wang, "Accelerating application bulk synchronous writes in hpc environments," in *Proceedings of the Seventh International Workshop on Systems and Network Telemetry and Analytics*, ser. SNTA '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 7–14. [Online]. Available: https://doi.org/10.1145/3660320.3660334

[29] R. I. S. Khan, A. H. Yazdani, Y. Fu, A. K. Paul, B. Ji, X. Jian, Y. Cheng, and A. R. Butt, "SHADE: Enable fundamental cacheability for distributed deep learning training," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*. Santa Clara, CA: USENIX Association, Feb. 2023, pp. 135–152. [Online]. Available: https://www.usenix.org/conference/fast23/presentation/khan

[30] A. Khan, A. K. Paul, C. Zimmer, S. Oral, S. Dash, S. Atchley, and F. Wang, "Hvac: Removing i/o bottleneck for large-scale deep learning applications," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 2022, pp. 324–335.

[31] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning i/o," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476181

[32] Redis Core Team, "Redis," Redis Labs, 2025. [Online]. Available: https://redis.io/

[33] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, p. 5, Aug. 2004.

[34] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: a heterogeneous-aware multi-tiered distributed i/o buffering system," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 219–230.

[35] S. McAllister, B. Berg, J. Tutuncu-Macias, J. Yang, S. Gunasekar, J. Lu, D. S. Berger, N. Beckmann, and G. R. Ganger, "Kangaroo: Caching billions of tiny objects on flash," ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 243–262. [Online]. Available: https://doi.org/10.1145/3477132.3483568

[36] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane dc persistent memory module," 2019. [Online]. Available: https://arxiv.org/abs/1903.05714

[37] "Andes – oak ridge leadership computing facility," https://www.olcf.ornl.gov/olcf-resources/compute-systems/andes/, Oak Ridge Leadership Computing Facility, 2025.

[38] M. Chai, E. Herron, E. Cervantes, and T. Ghosal, "Exploring scientific hypothesis generation with mamba," in *Proceedings of the 1st Workshop on NLP for Science (NLP4Science)*, L. Peled-Cohen, N. Calderon, S. Lissak, and R. Reichart, Eds. Miami, FL, USA: Association for Computational Linguistics, Nov. 2024, pp. 197–207. [Online]. Available: https://aclanthology.org/2024.nlp4science-1.17/

[39] Z. Hou, W. Gao, Y. Shen, F. Wang, and X. Liu, "Protransformer: robustify transformers via plug-and-play paradigm," in *Proceedings of the 38th International Conference on Neural Information Processing Systems*, ser. NIPS '24. Red Hook, NY, USA: Curran Associates Inc., 2025.

[40] B. Barker, "Message passing interface (mpi)," in *Workshop: high performance computing on stampede*, vol. 262. Cornell University Publisher Houston, TX, USA, 2015.

[41] Advanced Micro Devices, Inc., *RCCL Documentation*, 2025, accessed: 2025-08-08. [Online]. Available: https://rocmdocs.amd.com/projects/rccl/en/latest/

[42] NVIDIA Corporation, "Nccl: Nvidia collective communication library," https://developer.nvidia.com/nccl, 2024.

[43] A. Mathuriya *et al.*, "Cosmoflow: using deep learning to learn the universe at scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2019. [Online]. Available: https://doi.org/10.1109/SC.2018.00068

[44] TensorFlow Authors, "Tensorboard profiler," https://www.tensorflow.org/tensorboard/tensorboard_profiling_keras, 2021.

[45] M. N. Newaz and M. A. Mollah, "Memory usage prediction of hpc workloads using feature engineering and machine learning," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 2023, pp. 64–74.

[46] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on hpc systems," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2020, pp. 183–190.

[47] I. Peng, I. Karlin, M. Gokhale, K. Shoga, M. Legendre, and T. Gamblin, "A holistic view of memory utilization on hpc systems: Current and future trends," in *Proceedings of the International Symposium on Memory Systems*, 2021, pp. 1–11.

[48] K. V. Jha, S. Pandey, M. Annavaram, and A. Basu, "Hycache: Hybrid caching for accelerating dnn input preprocessing pipelines," in *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, 2025, pp. 433–448.

[49] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and mitigating data stalls in dnn training," *Proc. VLDB Endow.*, vol. 14, no. 5, p. 771–784, Jan. 2021. [Online]. Available: https://doi.org/10.14778/3446095.3446100

[50] W. Chen, S. He, Y. Xu, X. Zhang, S. Yang, S. Hu, X.-H. Sun, and G. Chen, "icache: An importance-sampling-informed cache for accelerating i/o-bound dnn model training," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 220–232.

[51] Y. Li, T. Wu, G. Li, Y. Song, and S. Yin, "Portus: Efficient dnn checkpointing to persistent memory with zero-copy," in *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, 2024, pp. 59–70.

[52] Y. Gu, A. Khadem, S. Umesh, N. Liang, X. Servot, O. Mutlu, R. Iyer, and R. Das, "Pim is all you need: A cxl-enabled gpu-free system for large language model inference," ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 862–881. [Online]. Available: https://doi.org/10.1145/3676641.3716267

[53] Y. Tang, P. Zhou, W. Zhang, H. Hu, Q. Yang, H. Xiang, T. Liu, J. Shan, R. Huang, C. Zhao, C. Chen, H. Zhang, F. Liu, S. Zhang, X. Ding, and J. Chen, "Exploring performance and cost optimization with asic-based cxl memory," in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 818–833. [Online]. Available: https://doi.org/10.1145/3627703.3650061

[54] H. Khetawat, N. Jain, A. Bhatele, and F. Mueller, "Predicting gpudirect benefits for hpc workloads," in *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2024, pp. 88–97.

[55] J. Y. Choi, M. Lupo Pasini, P. Zhang, K. Mehta, F. Liu, J. Bae, and K. Ibrahim, "Ddstore: Distributed data store for scalable training of graph neural networks on large atomistic modeling datasets," in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 941–950. [Online]. Available: https://doi.org/10.1145/3624062.3624171

[56] W. Shin, C. D. Brumgard, B. Xie, S. S. Vazhkudai, D. Ghoshal, S. Oral, and L. Ramakrishnan, "Data jockey: Automatic data management for hpc multi-tiered storage systems," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 511–522.

[57] L. Wang, S. Ye, B. Yang, Y. Lu, H. Zhang, S. Yan, and Q. Luo, "Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training," in *Proceedings of the 49th International Conference on Parallel Processing*, ser. ICPP '20. New York, NY, USA: Association for Computing Machinery, 2020.

[58] L. Kong, F. Mei, C. Zhu, W. Cheng, and L. Zeng, "Deepfetch: A node-aware greedy fetch system for distributed cache of deep learning applications," in *2024 International Conference on Networking, Architecture and Storage (NAS)*, 2024, pp. 1–8.

[59] A. V. Kumar and M. Sivathanu, "Quiver: an informed storage cache for deep learning," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, ser. FAST'20, 2020, p. 283–296.

[60] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning i/o," ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021.

[61] Z. Zhang, L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney, "Fanstore: Enabling efficient and scalable I/O for distributed deep learning," *CoRR*, vol. abs/1809.10799, 2018. [Online]. Available: http://arxiv.org/abs/1809.10799

[62] ThinkParQ GmbH, "BeeGFS - The Parallel Cluster File System," https://www.beegfs.io, 2024.

[63] P. J. Braam, "The lustre storage architecture," in *Proceedings of the 2003 Conference on Linux Symposium*, 2003, pp. 1–11. [Online]. Available: https://www.lustre.org/documentation/lustre-architecture.pdf

[64] B. Mao, S. Wu, and L. Duan, "Improving the ssd performance by exploiting request characteristics and internal parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 2, pp. 472–484, 2018.

[65] M. Farrell *et al.*, "MLPerf HPC: A Holistic Benchmark Suite for Scientific Machine Learning on HPC Systems," in *Proceedings of the IEEE/ACM Workshop on Machine Learning in High-Performance Computing Environments (MLHPC)*, 2021, pp. 33–45.

[66] D.-T. Nguyen, A. Bhattacharjee, A. Moitra, and P. Panda, "Deepcam: A fully cam-based inference accelerator with variable hash lengths for energy-efficient deep neural networks," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[67] M. F. Wehner, K. A. Reed, F. Li, Prabhat, J. Bacmeister, C.-T. Chen, C. Paciorek, P. J. Gleckler, K. R. Sperber, W. D. Collins *et al.*, "The effect of horizontal resolution on simulation quality in the c ommunity a tmospheric m odel, cam 5.1," *Journal of Advances in Modeling Earth Systems*, vol. 6, no. 4, pp. 980–997, 2014.

[68] Prabhat, S. Byna, V. Vishwanath, E. Dart, M. Wehner, and W. D. Collins, "Teca: Petascale pattern recognition for climate science," in *International Conference on Computer Analysis of Images and Patterns*. Springer, 2015, pp. 426–436.

[69] G. Haas and V. Leis, "What modern nvme storage can do, and how to exploit it: High-performance i/o for high-performance storage engines," *Proc. VLDB Endow.*, vol. 16, no. 9, p. 2090–2102, May 2023.

[70] D. Technologies. Ready solutions for hpc: Beegfs high-performance storage hdr100 refresh. [Online]. Available: https://infohub.delltechnologies.com/zh-cn/p/ready-solutions-for-hpc-beegfs-high-performance-storage-hdr100-refresh/

[71] EOFS, "Benchmarking lustre – setting realistic performance," 2024. [Online]. Available: https://www.eofs.eu/wp-content/uploads/2024/02/2.2-benchmarking_lustre.pdf

Appendix: Artifact Description

Artifact Description (AD)

VI. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 We contribute FitCache, a transparent drop-in multi-tier caching framework that leverages node-local memory (e.g., DRAM and PMem) and NVMe to accelerate distributed deep learning workloads on HPC systems, without requiring modifications to existing training pipelines or parallel file systems.

C_2 We develop a concurrent multi-tier fetching mechanism that issues parallel I/O requests across heterogeneous storage tiers and serves data from the fastest responder, eliminating centralized metadata dependencies and reducing data access latency under dynamic and heterogeneous runtime conditions.

C_3 We implement FitCache as a client–server system using LD_PRELOAD I/O interception and evaluate it on both large-scale and small-scale HPC platforms. Experiments on the Frontier supercomputer show up to 40% reduction in end-to-end training time and up to 71.6% reduction in per-batch I/O latency compared to the Lustre Orion PFS, while consistently outperforming NVMe-based caching baselines.

B. Computational Artifacts

A_1 https://github.com/akhanaornl/hvac (Baseline HVAC approach)

A_2 https://github.com/Garson-hu/FitCache.git (FitCache approach)

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_3	Figure 5-7
A_2	C_1, C_2, C_3	Figures 1, 3-9

VII. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

FitCache, built on top of POSIX-compatible I/O interfaces, introduces a transparent multi-tier caching design to accelerate distributed deep learning workloads. HVAC is the closest and most performant POSIX-compliant caching system for distributed DL training and therefore serves as the primary baseline for FitCache. FitCache adopts a client-server architecture with concurrent multi-tier data fetching to minimize data access latency without relying on centralized metadata services. The artifact supports contributions C_1, C_2, and C_3, enabling reproduction of experiments that demonstrate the effectiveness of FitCache in reducing training time in large-scale distributed deep learning scenarios.

Expected Results

This artifact supports the replication of the following key experimental results:

- Evaluation of FitCache shows up to 40% reduction in end-to-end training time compared to PFS-based baselines, and consistent performance improvements over NVMe-based HVAC across different scales.
- Per-batch I/O time is reduced by up to 71.6% compared to Lustre Orion PFS, and by more than 50% compared to NVMe-based HVAC under large-scale distributed training settings.
- FitCache demonstrates improved scalability under increasing GPU counts by mitigating backend storage contention through memory-tier caching and concurrent multi-tier data fetching.

Expected Reproduction Time (in Minutes)

The expected time to reproduce the artifact on a system similar to Frontier is as follows:

- Artifact Setup: Approximately 2250 minutes, including 120 minutes for experimental setup and about 2100 minutes for downloading the 1.3 TB CosmoFlow dataset and 2.4 TB DeepCam Dataset (assuming a download speed of 20 MB/s).
- Artifact Execution: Near 120 minutes, depending on experiment scale (number of GPUs, batch size, and so on).
- Artifact Analysis: Approximately 30 minutes to check runtime logs and calculate results.

Artifact Setup (incl. Inputs)

Hardware: The experiments were conducted on Frontier and an institution level cluster called ARC (GPUCLUSTER). Frontier is a supercomputer based on the HPE Cray EX architecture deployed at the Oak Ridge Leadership Computing Facility (OLCF). Frontier consists of 9,472 compute nodes, each equipped with one AMD EPYC 8124P CPU (64 cores, with two hardware threads per core) and eight AMD MI250X GPUs. Each node provides 512 GB of system memory and is provisioned with two node-local NVMe drives (1.92 TB each).

ARC (GPUCLUSTER) consists of 80 compute nodes and is used for small-scale evaluation. The node we used is equipped with one AMD EPYC 8124P CPU with 16 cores and one NVIDIA RTX 4060 Ti GPU with 16 GB memory. The cluster provides node-local storage and includes two nodes equipped with 512 GB of Persistent Memory, which are used to evaluate memory-tier caching behavior in FitCache.

Software: The FitCache implementation is written in C/C++ and requires the following dependencies:

- Mercury HPC communication library (v2.0.1)
- Log4c logging module (v1.2.4)
- GCC (v12.2.0)
- CMake (v3.28.3)

- Python (v3.10.13)

The Conda environment for our experiments is provided in the included `environment.yml` file. This environment ensures reproducibility of Python dependencies. To set up the environment, use:

```
conda env create -f environment.yml
conda activate fitcache_envs
```

FITCACHE requires the use of the `LD_PRELOAD`, as it operates in conjunction with deep learning applications that rely on NVMe devices or alternative storage media such as tmpfs installed in DRAM. This setup ensures that the caching hierarchy transparently intercepts I/O requests at runtime.

For the most up-to-date setup and execution instructions, please refer to the latest version of the FITCACHE GitHub repository provided with the artifact.

Datasets / Inputs: The primary dataset used in our experiments is the CosmoFlow and DeepCAM dataset, with a total size of approximately 1.3 TB and 2.4 TB. Before running the experiments, the dataset must be preloaded onto the parallel file system (PFS) to ensure efficient access across nodes.

Detailed dataset information, download instructions, and preprocessing scripts are available in the official CosmoFlow benchmark repository (https://github.com/sparticlesteve/cosmoflow-benchmark) and the official DeepCam benchmark repository (https://github.com/mlcommons/hpc_results_v3.0/tree/main/NVIDIA/benchmarks/deepcam/implementations/pytorch).

Installation and Deployment: To install and deploy FITCACHE in your system. Please follow the instructions and steps:

1) **Allocate compute resources:** If you use Slurm to manage the cluster, you can use commands like the following to get the needed resources for training:
```
salloc -t $TIME -p batch -N $K -C nvme
```

2) **Download FITCACHE:** After you have the needed resources, you can download the FITCACHE code through the GitHub link given above:
```
git clone https://github.com/Garson-hu/FitCache.git
```

3) **Build FITCACHE:** Go to the FITCACHE directory, and export the necessary environment variables:
```
export FITCACHE_SERVER_COUNT=4
export FITCACHE_LOG_LEVEL=800
export RDMAV_FORK_SAFE=1
export BBPATH=YOUR_FASTEST_CACHE_TIER
export FITCACHE_DATA_DIR=YOUR_DATA_PATH
```

   After that, create the `build` directory
```
mkdir -p build && cd build
./build.sh
```

4) **Start the server:**
```
./src/fitcache_server FITCACHE_SERVER_COUNT &
```

5) **Case 1: Run DL application with FITCACHE:**
```
LD_PRELOAD=./src/libfitcache_client.so
python train_model.py
```

6) **Case 2: Run an example with FITCACHE:**
```
LD_PRELOAD=./src/libfitcache_client.so
tests/basic_test
```