Fault-Tolerant Deep Learning Cache with Hash Ring for Load Balancing in HPC Systems

Seoyeong Lee*, Awais Khan†, Yoochan Kim*, Junghwan Park*, Soon Hwang*, Jae-Kook Lee‡, Taeyoung Hong‡, Christopher Zimmer†, Youngjae Kim*
*Sogang University, South Korea, †Oak Ridge National Laboratory, US, ‡KISTI, South Korea

***Abstract*—Large-scale DL on HPC systems like Frontier and Summit uses distributed node-local caching to address scalability and performance challenges. However, as these systems grow more complex, the risk of node failures increases, and current caching approaches lack fault tolerance, jeopardizing large-scale training jobs. We analyzed six months of SLURM job logs from Frontier and found that over 30% of jobs failed after an average of 75 minutes. To address this, we propose fault-tolerance strategies that recache data lost from failed nodes using a hash ring technique for balanced data recaching in the distributed node-local caching, reducing reliance on the PFS. Our extensive evaluations on Frontier showed that the hash ring-based recaching approach reduced training time by approximately 25% compared to the approach that redirects I/O to the PFS after node failures and demonstrated effective load balancing of training data across nodes.**

***Index Terms*—HPC, Distributed Deep Learning, NVMe Cache, Fault Tolerance**

I. Introduction

Large-scale distributed deep learning training can consume a significant portion of time—up to 60-70%—on I/O operations in HPC environments [1]. This considerable I/O overhead is not only due to the traditional parallel file system (PFS) relying on slow hard disk drives but also stems from inherent I/O bottlenecks in the PFS metadata server, which struggles with the simultaneous access demands of numerous small files. To address this bottleneck, extensive research has explored the use of node-local NVMe SSDs for caching [2]–[11]. Among these solutions, HVAC (High-Velocity AI Cache) [11] represents a state-of-the-art distributed caching system for large-scale AI training jobs by leveraging node-local NVMe storage on compute nodes to cache large training datasets. HVAC has demonstrated its effectiveness on Frontier [12], the world's fastest supercomputer.

HVAC employs a distributed caching system, where data is distributed and cached across all compute nodes, allowing each node to access data from its own NVMe or from NVMes on remote nodes. This approach effectively reduces data retrieval times and enhances the overall performance of DL tasks by avoiding access to the shared PFS and minimizing metadata overhead. However, such distributed caching systems, specifically for AI workloads, remain vulnerable to node failures. As the number of compute nodes increases in DL, the probability of node failure increases correspondingly. Considering the extended training times in DL, this issue becomes non-negligible. Therefore, we performed a job failure analysis in Frontier [12] to better understand this problem; for more details, see Section III.

In conventional DL systems, the primary concern during a node failure is the loss of model parameters or the computational state. Therefore, previous study efforts [13]–[18] have been directed toward preserving model state and accelerating its recovery in the event of failures. For instance, the resilience strategies [16] proposed immediate mini-batch rollback and lossy forward recovery, aim to minimize the runtime impact by continuing training without restarting from checkpoints. FastPersist [17] introduced NVMe optimizations for faster checkpoint writes to SSDs and implemented overlapping of checkpointing with computations to minimize checkpointing overhead. Elastic Deep Learning [14] proposed an elastic scaling method utilizing User-Level Failure Mitigation (ULFM) in MPI, thereby reducing recovery time in volatile computational environments.

However, node failures in distributed caching systems introduce additional challenges that these methods do not address. The learning process becomes particularly vulnerable to interruption if the underlying caching system, such as HVAC, lacks robust fault tolerance mechanisms. This vulnerability stems from the caching system's critical role in managing data flow—a failure at this level can disrupt the entire training process. Consequently, without fault-tolerant caching solutions, the benefits of the fault tolerance implemented within the DL framework may be compromised.

Moreover, node failures in caching systems result in the loss of data cached on the failed node's NVMe storage, which is essential for subsequent training epochs. If cached data lost due to node failure in the system is not restored, reliance on continuous access to the PFS could potentially extend training time. This is particularly problematic in scenarios involving vast datasets or early-stage failures. Intermittent PFS access by training nodes increases I/O latency, introducing temporal inconsistencies in learning processes across distributed nodes. This asynchronous behavior worsens the straggler problem, potentially triggering cascading failures and violating job time limits. In addition, utilizing the existing hash functions adopted in the distributed deep learning caching system to redistribute data among the remaining $N-1$ nodes, while excluding dead nodes, necessitates extensive data migration between the accessible nodes-a process that incurs significant computational and time costs.

To address these problems, we propose a fault-tolerant HVAC system. Our contributions are as follows:

- We conducted an in-depth analysis of job failures over a six-month period on the Frontier system, revealing that failures

could be critical to the HVAC system.
- We designed a fault-tolerant system on HVAC, equipped with (i) a data recaching mechanism within the HVAC layer to ensure data availability and fast access in the event of a node failure, and (ii) a hash ring technique for load-balanced data recaching in HVAC.
- Lastly, we implemented fault-tolerant system on HVAC by extending the HVAC system. Our extensive evaluations on the Frontier supercomputer demonstrated that our Hash ring-based recaching approach led to an 24.9% reduction in runtime on 1024 nodes compared to a baseline approach of retrieving lost data from PFS in the event of node failure.

II. Background

A. I/O Challenges in Distributed Deep Learning

DL applications have unique I/O characteristics that significantly impact HPC performance. A key factor is the repetitive data access, as DL models, particularly in vision tasks, read the same dataset across multiple training epochs, highlighting the need for efficient data locality. Additionally, data shuffling is crucial for improving model generalization and reducing overfitting. In data parallelism, each node reads a dataset portion, and subsequent epochs involve shuffling, requiring random access to different data segments. This process intensifies I/O demands, especially in distributed DL environments with concurrent compute nodes, necessitating a robust data access strategy.

The use of PFS in DL environments faces significant challenges due to centralized metadata servers, which can become bottlenecks under heavy access loads. Metadata lock contention arises when multiple processes access metadata simultaneously, causing delays as operations are serialized to ensure consistency. DL training datasets, often composed of many small files, generate high volumes of metadata I/O requests, exacerbating the bottleneck. Traditional PFS, optimized for large sequential accesses, struggles with the small, random accesses typical in DL workloads, hindering performance and scalability. This centralized metadata management approach fails to scale effectively with growing datasets and compute nodes, limiting overall system throughput.

B. High-Velocity AI Cache (HVAC)

The HVAC system [11] introduced a node-local NVMe cache layer specifically designed for distributed DL. The idea is to minimize both metadata and data I/O overhead during PFS data access operations. The system achieves this optimization by strategically distributing the training dataset across node-local NVMe SSDs of compute nodes, effectively leveraging the repetitive I/O patterns of DL workloads.

HVAC consists of a client and a server, with each instance running on every compute node. The HVAC server, running as a separate daemon from the DL application, handles actual data I/O operations. The HVAC client, a shared library attached to the DL application, intercepts data open/read/close system calls. Each file is mapped to a specific server based on a hash function. By running the hash function, the client determines which node caches the required data and sends an RPC request to the corresponding server. Upon receiving this RPC, the HVAC server first checks its NVMe for cached data. If the data is cached, it performs I/O from the NVMe and transmits to the client; if not, it reads from the PFS and transmits, while instructing a data mover thread to copy this data to the NVMe for future use.

III. Job Failures on the Frontier HPC System

To gain a comprehensive understanding of job failures in Frontier supercomputer, we analyzed *Slurm* data over a period of six months since the system's production launch. Our analysis focused on jobs marked with the states *Job Fail*, *Node Fail*, and *Timeout*, excluding those canceled by users, system administrators, or during maintenance. Each failure scenario is defined as follows.

- *Job Fail* results from code errors, data issues, environment problems, or external system malfunctions.
- *Node Fail* occurs when a specific node stops functioning due to hardware issues, network problems, software bugs, or overload, disrupting job execution.
- *Timeout* happens when a job does not complete within a set time limit, often due to job complexity, resource shortages, or network latency.

TABLE I
Analysis of job failures on Frontier supercomputer over the period of six months since in production.

Type	Count	Failure ratio	Overall ratio
Total Jobs	181,933	N/A	100%
Total Failures	45,556	100%	25.04%
Node Fail	1,174	2.58%	0.65%
Timeout	20,464	44.92%	11.25%
Job Fail	23,918	52.50%	13.15%

Table I shows the analysis of job failures on the Frontier supercomputer over six months of production. Of 181,933 jobs, 45,556 (25.04%) failed. Failures were classified as "Job Fail" (52.50%), "Timeout" (44.92%), and "Node failure" (2. 58%). In this study, we define node failures to include both "Node Fail" and "Timeout" cases, which together account for about half of all failures in this period. This is because, in both cases, the node becomes unresponsive or fails to perform its designated tasks, disrupting system operations. Specifically, a timeout is considered a node failure because, in our context, it primarily refers to a network timeout. This occurs when a node does not respond within a specified time frame, often due to network issues such as router or switch failures, high latency, or packet loss.

Figure 1 illustrates the average elapsed time of failed jobs on the Frontier supercomputer. The data indicates that, on average, jobs run for over an hour before failing. Notably, in some weeks, particularly for "Node Fail" and "Timeout" cases, the elapsed time before failure reaches two to three hours. This is especially detrimental, as long-running jobs that terminate abruptly represent significant losses in computational resources and time. Furthermore, the data shows that job failures occur consistently every week, highlighting a persistent issue.

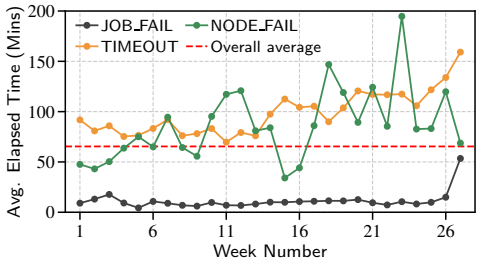
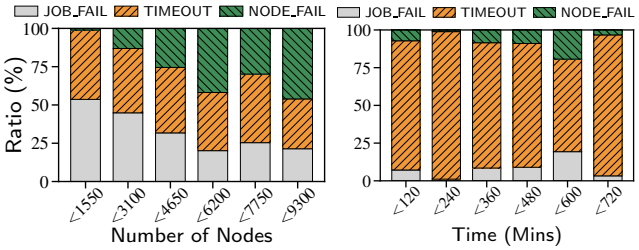


Fig. 1. Analysis of job failures over time on the Frontier system, showing the average elapsed time in minutes per week for 27 weeks. The graph categorizes jobs into *JOB_FAIL*, *TIMEOUT*, and *NODE_FAIL*, with the overall average elapsed time represented by the red dash line.



(a) Node Scaling Impact (b) Runtime Impact

Fig. 2. Distribution of job failure types on the Frontier supercomputer over the period of six months since in production, categorized by (a) node count and (b) elapsed time.

Further analysis is presented in Figure 2, which provides deeper insights into the relationship between job failures and system parameters. Figure 2(a) illustrates the distribution of types of failure in jobs relative to the number of nodes involved. The data reveal a clear trend: as the node count increases, the proportion of "Node Fail" incidents rises, indicating that hardware failures become more frequent as more nodes are utilized. Specifically, within the node range of 7,750 to 9,300, "Node Fail" incidents account for 46.04% of failures. When combined with "Timeout" incidents, these two categories constitute 78.60% of the failure statistics in this range.

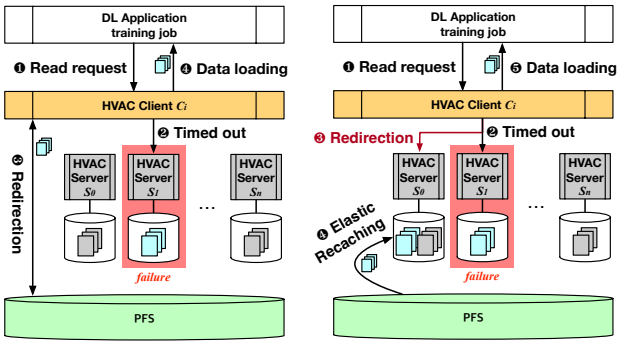
Figure 2(b) examines the distribution of job failures in relation to the elapsed time before failure. The data suggest that the duration of runtime does not significantly affect the ratio of failure types. This implies that while jobs may run for extended periods, the likelihood of failure is closely tied to the number of nodes, with failure events potentially occurring at any time. This finding underscores the importance of node count as a critical factor in hardware failures.

The implications of these results are profound, particularly for software systems not designed to handle such failures gracefully. For instance, a failure in HVAC during execution without a recovery mechanism could necessitate restarting the entire training process, potentially requiring access to several terabytes of data again.

IV. Design of Fault-Tolerant DL Cache

A. Addressing Failure via IO Redirection to PFS

To achieve fault tolerance in a distributed caching system for DL, a straightforward approach is to redirect I/O requests



(a) PFS redirection approach (b) Elastic recaching with hash ring

Fig. 3. Overview of fault-tolerance designs for HVAC systems: (a) PFS redirection approach and (b) Elastic recaching with hash ring.

intended for a failed node's HVAC server to the PFS. This method leverages the fact that training data is stored in the PFS, allowing access to data lost from the caching layer due to failure. This approach operates in two key stages post-failure: fault detection and I/O redirection. The system can use a timeout-based logic for fault detection.

Figure 3(a) illustrates the PFS redirection sequence. Each HVAC client tracks active and faulty nodes, monitoring for timeouts on each request. Upon a timeout, the client increments a counter and redirects the request to the PFS. The timeout counter is implemented to mitigate the risk of false positives, ensuring that transient network delays do not prematurely trigger error handling procedures. Once the timeout count for a specific node reaches a predefined threshold, that node is flagged as failed. The operational flow proceeds as follows: ❶ Upon receiving an I/O read request, the HVAC client C_i intercepts this request via LD_PRELOAD. ❷ If the RPC request to HVAC Server S_1 experiences multiple timeouts, it is designated as failed. ❸ Consequently, the I/O request is redirected to the PFS. ❹ Upon receipt of the data from the PFS, C_i returns it to the training job.

This design allows each node to autonomously detect failures, thereby eliminating the need for additional inter-node communication. Subsequent requests are directed to the PFS without attempting to access the failed node. This approach ensures system resilience by maintaining data availability even in the event of node failures. Although timeout detection is performed by every node, the delay incurred is tolerable. The Time-to-Live (TTL) parameter only needs to be greater than the longest observed latency, which indicates that the lower bound—the longest latency—is not excessively high. Therefore, determining an appropriate TTL, considering this factor, does not significantly impact overall performance.

The PFS redirection approach is effective for small datasets due to the low absolute amount of data. It is also effective for late-epoch failures, where the number of remaining epochs is minimal. In these cases, the infrequent PFS accesses per epoch after a failure result in an acceptable performance impact. However, this approach faces significant challenges with larger datasets or early-epoch failures, leading to performance degradation. These scenarios necessitate frequent PFS accesses due to the substantial loss of cached data or the prolonged

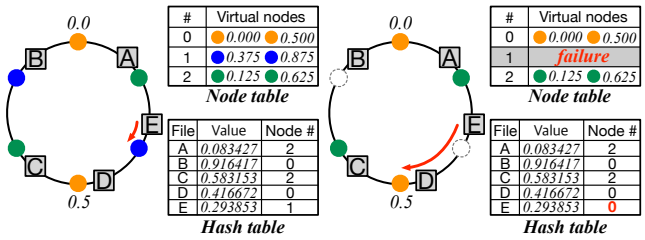
absence of caching effects. The increased PFS access latency in these situations can lead to two significant issues, further compounding the challenges faced by the system.

1) ***Straggler problem****:* As the number of nodes increases, a severe straggler problem emerges. This is due to the batch-wise synchronization characteristic of DL. When a small number of nodes experience delays in batch processing due to PFS access, the majority of nodes must wait for these slower nodes. This batch synchronization causes the straggler problem to occur with each batch, accumulating over time. Consequently, it significantly impairs parallelism and scalability, substantially increasing the overall training time.

2) ***Job time limit violations****:* The prolongation of training time poses a substantial risk, as it can lead to job time limit violations in high-performance computing environments. In scenarios where a DL job is allocated a specific runtime (e.g., 2 hours), the redirection to PFS can cause an unexpected increase in execution time. This increase may seem minor in percentage terms, but its implications can be severe. Even a modest 5-10% increase in runtime could push the job beyond its allocated time slot, resulting in premature termination by the job scheduler. This not only wastes computational resources but also disrupts the training process, potentially leading to incomplete or invalid results. Moreover, in competitive production environments where multiple users share HPC resources, such overruns can have cascading effects on subsequent job schedules and overall system utilization. Therefore, while PFS redirection offers a straightforward solution to node failures, its impact on job runtime predictability presents a significant challenge for designing fault-tolerant DL systems.

B. Elastic Recaching with Hash Ring

In the original HVAC implementation, static hash partitioning distributes data uniformly across nodes by hashing the data path as a key, then applying a modulo operation with the number of nodes to the resulting hash value. This determines the node that stores the target data, allowing the HVAC client to send the RPC request for I/O to the appropriate node's HVAC server. However, when a node fails, the HVAC client must recalculate the hash based on the remaining nodes $N-1$ instead of N, affecting all hash calculations. This leads to significant data redistribution across the surviving nodes, resulting in substantial data movement. Not only is the lost data reassigned to other nodes, but well-cached data is also relocated. This process can be inefficient and time-consuming in large-scale systems. Employing multiple hash functions can also be considered. This technique involves using additional hash functions to redistribute data that resided on the failed node. While this strategy can reduce data movement by only redistributing the affected data, it still faces challenges, particularly in handling repeated node failures, which can introduce scalability issues. Another method, range partitioning [19], assigns data to nodes based on contiguous key ranges. In the event of a node failure, the data within the failed node's range is the first to be redistributed. However, maintaining load balance might require adjustments to other nodes' data ranges as well, leading to more extensive redistribution.



(a) Before fault (b) After fault

Fig. 4. Hash ring mechanism for fault tolerance in NVMe caching with multiple virtual nodes. (a) Before fault: Initial data distribution. (b) After fault: Data is reassigned to the next closest node clockwise after a node failure.

To effectively recache the lost data, we implemented a consistent hashing method known as hash ring [20]. In this approach, both data items and nodes are mapped to positions on a logical circular ring using a hash function. Each data item is assigned to the node with the nearest hash value in the clockwise direction on the ring. Upon node failure, only the data items previously assigned to the failed node require reassignment to the next nearest node in the clockwise direction, thus minimizing data movement.

Fig. 4 illustrates the fault tolerance mechanism in NVMe caching using a hash ring. In part (a), data items are initially distributed to nodes based on their hash values in a clockwise manner. For example, file E with a hash value of 0.293853 is assigned to Node 1. In part (b), the system after Node 1 fails. Data previously managed by Node 1, such as file E, is reassigned to the next clockwise node, Node 0. This reassignment strategy ensures only the absolute theoretical minimum data movement by redistributing only the data from the failed node to its adjacent node.

However, this mechanism may lead to load imbalance issues, as one node might receive a disproportionate amount of data. To address this problem, the concept of virtual nodes is implemented. Each physical node is represented by multiple points on the hash ring, distributing its range of responsibility more evenly, thereby addressing the load imbalance. As a result, the data is spread more evenly among nodes, reducing the likelihood of any single node receiving a disproportionate amount of data. Increasing virtual nodes allows for more even data distribution, mitigating load imbalance. However, this creates a trade-off with increased resource overhead, such as memory usage and computational costs. Thus, carefully balancing these factors is critical for optimal system performance.

Based on these principles, we have implemented an elastic recaching approach, as illustrated in Figure 3(b). The process begins with clients constructing a hash ring during initialization, based on the number of server nodes and a predefined number of virtual nodes. ❶ The DL application's I/O read request is intercepted by the HVAC client. File paths serve as hash keys to determine the appropriate server for data I/O requests. The HVAC client sends an RPC request to the corresponding HVAC server, S_1. ❷ Upon detecting a timeout, the client determines that the server has failed and removes the faulty node from the hash ring. The approach handles node failures by reassigning data responsibilities across the

remaining nodes elastically to ensure continuous operation. ❸ The request and subsequent requests for lost data are redirected to newly assigned nodes based on the updated hash ring. When a new owner server, S_0, receives a request, it checks its NVMe for the requested file. ❹ If the file is present, the server serves it directly to the client. Otherwise, as this represents the first epoch after the failure where the lost files are not yet cached, the server follows a three-step process: retrieving the file from the PFS, serving it to the client, and then caching it locally. ❺ Upon receiving the response from the RPC call, the client returns the requested data contents to the application.

This approach results in only one additional PFS access per lost data item, significantly reducing the performance impact of node failures compared to continuous PFS redirection.

Upon node failure, the system efficiently redistributes data by removing the failed node from the hash ring and automatically redirecting its data to the next nearest virtual node. The implementation employs map data structure to facilitating efficient data retrieval and reassignment during node failures. The logarithmic time complexity of map operations enables swift adaptation to node failures with minimal overhead. This strategy minimizes data movement, as only the lost data from the failed node requires repopulation from the PFS to the NVMe storage of the new node.

V. Evaluation

A. Experimental Setup

We used Frontier supercomputer for large-scale evaluation of the proposed system. In Frontier, each node contains two 1.9 TB Samsung PM9A3 M.2 NVMe SSDs for node-local storage, and the details of each node are shown in Table II. The node-local SSDs are aggregated into a single volume presented as an XFS file system using a RAID0 128 KiB stripe configuration. Therefore, each compute node provides 3.5 TB of usable capacity with roughly 4 GB/s of peak sequential write and 8 GB/s of peak sequential read bandwidth. The Frontier runs on a center-wide shared Lustre parallel file system named Orion.

1) Implementation: We extended the existing HVAC system to support fault tolerance features. HVAC was originally built using C++ and the Mercury RPC communication library [21]. We implemented Hash ring with the *std::map* class from C++ STL. This implementation involved the modification and extension of 1k lines of code.

2) Application and Dataset: Cosmoflow [22] is a highly scalable DL application from the MLPerf HPC v0.5 benchmark suite. It involves training a 3D convolutional neural network on N-body cosmology simulation data to predict the physical parameters of the universe.

TABLE II
The computer node specifications of Frontier.

Attribute	Description
Supercomputer	Frontier
CPU	AMD Trento EPYC 7A53
GPU	8 x MI250X AMD with 64 GiB HBM
Memory Capacity	512 GiB DDR4
Node-local Storage	2 x 1.9 TB Samsung PM9A3 M.2 NVMe

CosmoFlow is built on DL framework Horovod. We run the application with Horovod elastic run to support elastic training [13], a feature that allows adjusting the number of participating nodes during runtime training. With Elastic training, CosmoFlow can continue training even in the event of node failure by reverting to the start of the failed epoch.

We used the cosmoUniverse dataset containing preprocessed 1.3TB TFRecord files generated from simulations run by the ExaLearn group at NERSC. There are 524,288 samples for training and 65,536 samples for validation. All the dataset is stored on the Orion file system before any training run. We ran 5 epochs per experiment and all experiments were repeated three times unless otherwise noted.

3) Injecting Random Failures: In our experiments, node failures were randomly injected after the completion of the first epoch to simulate fault-tolerance scenarios. This ensures that all data is cached before the failure event, allowing us to assess the impact of lost cached data. The failures were introduced by disabling one or more nodes during runtime, simulating unexpected failures. Specifically, we implemented this by employing a *SLURM sacct update NodeName=node_name State=DRAIN* that can disable or isolate a node from the rest of the allocated nodes at a predefined or random point in time after the first epoch. This method effectively simulates controlled failures in HPC environments, making the node unusable. To avoid bias, both the timing and node selection were randomized.

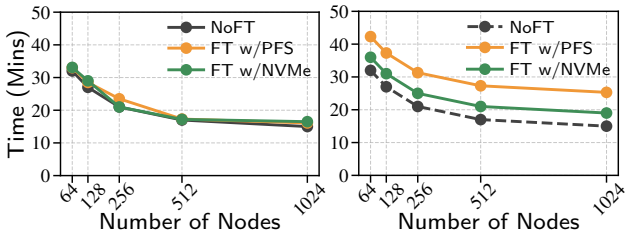
We compare the following systems:

- **NoFT**: The baseline HVAC [11] without any fault-tolerance support.
- **FT w/ PFS (Sec IV-A)**: The fault-tolerant version of HVAC redirects IO to PFS every time the files are not available in the HVAC cache.
- **FT w/ NVMe (Sec IV-B)**: The elastic recaching via Hash ring version of HVAC, which reactively caches the files upon missing from the cache as a result of failure. The virtual node count is set to 100 per physical node.

B. Evaluation Results

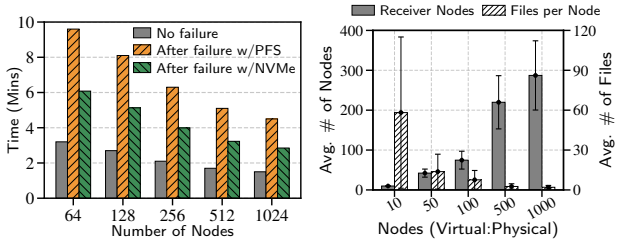
1) Overall Performance: In Figure 5(a), under non-failure conditions, all configurations demonstrate reduced operation time as the number of nodes increases, consistent with the anticipated performance improvements from increased parallelism. The overall trend in operation time is consistent across configurations, with variations of 1-2 minutes, within acceptable error margins. Notably, the NoFT configuration consistently achieves the best performance. This is attributed to the overhead introduced by fault tolerance mechanisms in the FT w/ PFS and FT w/ NVMe configurations. Specifically, these mechanisms require additional conditional checks, timeout monitoring, and mutex locks to manage and track the data structures associated with fault-tolerant operations, which slightly impacts their execution time.

In Figure 5(b), single-node failures occur randomly five times after the first epoch to ensure the cache is fully populated when node failure occurs. This setup allows us to observe the impact of losing cache during node failures. The dashed line



(a) Without failure (b) With 5 failures

Fig. 5. End-to-end training time of the three systems with and without failure during training. The dashed line represents the elapsed time of baseline system without failure.



(a) Victim epoch (b) Load Balancing

Fig. 6. (a) An in-depth analysis of the victim epoch during which a failure happened. (b) Impact of virtual node count on post-failure load redistribution: Error bars show the standard deviation.

represents the original no-failure case, shown as a dash because the baseline HVAC lacks fault-tolerant aspects, resulting in immediate job termination upon failure. FT w/PFS and FT w/NVMe show higher elapsed times than the no-fault scenario, as expected. This increase is attributed to the training process continuing with $N-1$, $N-2$, $\cdots$ nodes after failures, and the time required for Horovod elastic run, to roll back to the start of the epoch and resume training.

On the HVAC side, FT w/PFS shows the highest elapsed time, with a 32.2% increase at 64 nodes and a 68.7% increase at 1024 nodes compared to the non-failure case. This is due to continuous PFS access for lost data after failure, making its performance similar to the baseline HVAC's initial uncached epoch for the lost data. In contrast, FT w/NVMe exhibits only a 12.5% increase at 64 nodes and 26.7% at 1024 nodes, outperforming FT w/PFS by 14.8% and 24.9% at these respective scales. The advantage of FT w/NVMe lies in its recaching mechanism, which accesses the PFS only once after failure, avoiding repeated access in subsequent epochs.

As node count increases, both FT w/NVMe and FT w/PFS show growing performance degradation relative to their non-failure counterparts, with FT w/NVMe showing a 12.5% to 26.7% increase and FT w/PFS showing a 32.2% to 68.7% increase from 64 to 1024 nodes. This increasing overhead is attributed to the fixed time required for Horovod's elastic run resumption, which becomes more significant as baseline training time decreases with increased parallelism.

Despite this overall trend of increasing relative overhead, the gap between FT w/NVMe and FT w/PFS does not reduce as the number of nodes increases. Theoretically, the overhead due to node failure should decrease with more nodes because the data loss is reduced, and PFS access decreases. However, the results indicate that the straggler effect intensifies with more nodes, causing batch processing time to depend on the slowest nodes accessing the PFS. Although a smaller number of nodes access PFS and the majority of the nodes perform I/O quickly, the overall performance is still limited by the slowest nodes. This observation underscores the importance of minimizing PFS access in fault-tolerant DL, even as the system scales to larger node counts.

Figure 6(a) presents an analysis of the time per epoch in the event of a failure across systems ranging from 64 to 1024 nodes. The chart compares three scenarios: epochs without failures, epochs where PFS redirection was employed post-failure, and epochs utilizing the fault-tolerant NVMe recaching mechanism. Epochs without failures exhibit the shortest completion times. PFS redirection leads to significantly longer epoch durations, particularly at smaller scales (64–128 nodes), due to the overhead of slower PFS access. In contrast, the NVMe recaching mechanism achieves significantly better performance during failure scenarios, with times approaching those of the no-failure condition as the node count increases, demonstrating its scalability and effectiveness in larger distributed systems.

2) Load Distribution Analysis: Figure 6(b) presents the simulation results illustrating the effect of varying the number of virtual nodes per physical node on load distribution following a node failure in a distributed system. The simulation was conducted 500 times using 1024 physical nodes, each configured with a different number of virtual nodes. The plotted values represent the average outcomes across all trials. The left y-axis shows the average number of nodes receiving redistributed data, while the right y-axis indicates the average amount of files each node received.

As the number of virtual nodes per physical node increases, the average number of nodes absorbing redistributed data also rises, as shown by the upward trend in the"Receiver Nodes" bars. For instance, with 10 virtual nodes per physical node, a failure leads to the redistribution of data across only 3 physical nodes. However, this number approaches 300 nodes when the virtual-to-physical node ratio reaches 1000:1. As the virtual-to-physical node ratio increases, the growing standard deviation reflects significant variability in data distribution. The "Files per Node" metric decreases as more nodes participate in the redistribution process, resulting in a more balanced and consistent data distribution. The reduction in standard deviation with higher virtual node counts further confirms improved load balancing with respect to file distribution.

While the number of receiver nodes increases with a higher ratio of virtual nodes per physical node and the load imbalance improves, the "Receiver Nodes" metric does not scale linearly. As shown in the figure 6(b), the rate of increase in the "Receiver Nodes" metric declines significantly beyond 500 virtual nodes per physical node. Based on our observations, the average number of receiver nodes stabilizes around 350, even as the number of virtual nodes per physical node continues to rise, with only marginal increases thereafter. This indicates a point of diminishing returns, where further increasing the number of virtual nodes no longer significantly enhances data distribution. Additionally, increasing the number of virtual

nodes enlarges the hash table, which in turn heightens resource consumption and prolongs computational time. Therefore, there is a trade-off, and simply increasing the number of virtual nodes is not always beneficial. The optimal number of virtual nodes per physical node exists for each system and depends on the number of data files used. In our environment, this optimal number was 100.

VI. Conclusion

This paper presents a fault-tolerant caching system for large-scale deep learning in HPC environments. Our analysis of job failure logs from Frontier confirmed that node failures occur frequently. To support fault tolerance in HVAC, we implemented an elastic recaching mechanism based on a hash ring. The hash ring-based fault-tolerant HVAC system redistributes only the lost data across surviving nodes in a load-balanced manner, ensuring cache resilience with minimal performance impact, incurring only a single additional PFS access per failure event.

VII. Acknowledgement

This work was supported in part by the Korea Institute of Science and Technology Information (Grant No. K24L2M1C1) and by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. RS-2024-00416666). This research 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. The authors would also like to thank Professor Xubin He from Temple University for his valuable insights and contributions to discussions related to this paper. Youngjae Kim is the corresponding author.

References

[1] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning i/o," *arXiv preprint arXiv:2101.08734*, 2021.

[2] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Proceedings of the 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–11, 2012.

[3] G. Lockwood, D. Hazen, Q. Koziol, *et al.*, "Storage 2020: A vision for the future of hpc storage," Tech. Rep. LBNL-2001072, Lawrence Berkeley National Laboratory (LBNL), 2017.

[4] M. Hennecke, "Daos: A scale-out high performance storage stack for storage class memory," *Supercomputing Frontiers*, vol. 40, pp. 68–73, 2020.

[5] "Beeond (beegfs on demand)." http://www.beegfs.io/wiki/BeeOND. Accessed: 2024-07-24.

[6] "Encryption and a local read-only cache (lroc) device." https://www.ibm.com/docs/en/spectrum-scale/5.0.4?topic=encryption-local-read-only-cache-lroc-device. Accessed: 2024-07-24.

[7] "Unify-cr." https://github.com/LLNL/UnifyFS. Accessed: 2024-07-24.

[8] Y. Qian, X. Li, S. Ihara, A. Dilger, C. Thomaz, S. Wang, W. Cheng, C. Li, L. Zeng, and F. Wang, "Lpcc: Hierarchical persistent client caching for lustre," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, 2019.

[9] H. Devarajan, A. Kougkas, and X.-H. Sun, "Hfetch: Hierarchical data prefetching for scientific workflows in multi-tiered storage environments," in *Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 62–72, 2020.

[10] 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 (SC '21)*, 2021.

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

[12] "The frontier." https://www.olcf.ornl.gov/frontier/. Accessed: 2024-07-24.

[13] "Elastic training." https://horovod.readthedocs.io/en/latest/elastic_include.html. Accessed: 2024-07-24.

[14] J. Li, G. Bosilca, A. Bouteiller, and B. Nicolae, "Elastic deep learning through resilient collective operations," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pp. 44–50, 2023.

[15] B. Nicolae, J. Li, J. Wozniak, G. Bosilca, M. Dorier, and F. Cappello, "Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models," in *Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pp. 172–181, 2020.

[16] B. Nicolae, T. Hobson, O. Yildiz, T. Peterka, and D. Morozov, "Towards low-overhead resilience for data parallel deep learning," in *Proceedings of the 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 336–345, 2022.

[17] G. Wang, O. Ruwase, B. Xie, and Y. He, "Fastpersist: Accelerating model checkpointing in deep learning," in *Proceedings of the 2024 IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2024.

[18] Y. Kim, K. Kim, Y. Cho, J. Kim, A. Khan, K. Kang, B. An, M. Cha, H. Kim, and Y. Kim, "Deepvm: Integrating spot and on-demand vms for cost-efficient deep learning clusters in the cloud," in *Proceedings of the 2024 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2024.

[19] M. Özsu and P. Valduriez, *Principles of Distributed Database Systems*. Springer, 4th ed., 2020.

[20] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97)*, pp. 654–663, 1997.

[21] "Mercury: Rpc framework specifically designed for use in hpc systems." https://github.com/mercury-hpc/mercury. Accessed: 2024-07-24.

[22] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Karna, D. Moise, S. Pennycook, K. Maschhoff, J. Sewall, N. Kumar, S. Ho, M. Ringenburg, Prabhat, and V. Lee, "Cosmoflow: Using deep learning to learn the universe at scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*, IEEE Press, 2018.

Appendix: Artifact Description

Artifact Description (AD)

VIII. Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 We conducted an in-depth analysis of job failures over a six-month period on the Frontier system, revealing that failures could be critical to the HVAC system.
- C_2 We designed a fault-tolerant system on HVAC, equipped with (i) a data recaching mechanism within the HVAC layer to ensure data availability and fast access in the event of a node failure, and (ii) a hash ring technique for load-balanced data recaching in HVAC.
- C_3 We implemented the fault-tolerant system on HVAC and demonstrated through extensive evaluations on the Frontier supercomputer that our hash ring-based recaching approach led to a 24.9% reduction in runtime on 1024 nodes compared to a baseline approach of retrieving lost data from PFS in the event of node failure.

B. Computational Artifacts

- A_1 https://doi.org/10.5281/zenodo.13347306 (PFS Redirection Approach)
- A_2 https://doi.org/10.5281/zenodo.13347304 (Elastic Recaching Approach with Ring Hash)

The source code, setup instructions, and additional documentation for FT-Cache can be found in the GitHub repository: https://github.com/lass-lab/FT-Cache/.

TABLE III
Overview of Artifacts and their Contributions to Paper Elements

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_2	Figures 3, 5, 6(a)
A_2	C_2, C_3	Figures 3-6

IX. Artifact Identification

A. Computational Artifact A_1 and A_2

Relation To Contributions

FT-Cache, built on top of the HVAC system, introduces fault tolerance and load balancing through a hash ring-based recaching mechanism. The artifact supports contributions C_2, and C_3, enabling the reproduction of the experiments that demonstrate the effectiveness of FT-Cache in reducing runtime and improving load distribution in distributed DL scenarios, particularly under node failure conditions.

Expected Results

This artifact allows for the replication of the following key experimental results:

- Evaluation of FT-Cache's impact on runtime before and after node failures, demonstrating a runtime reduction of up to 24.9% compared to baseline methods relying on PFS.
- Simulational analysis of load distribution, showing improved balance with the implementation of virtual nodes within the hash ring.
- Examination of the efficiency of the fault detection and recaching process, which highlights the reduced overhead in restoring normal operations after node failures.

Expected Reproduction Time (in Minutes)

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

- Artifact Setup: Approximately 2 days and 120 minutes, including 120 minutes for experimental setup and two days for downloading the CosmoFlow dataset, as it is large.
- Artifact Execution: 120-180 minutes, depending on the scale of the experiments (number of nodes and epochs).
- Artifact Analysis: 60 minutes, focusing on the interpretation of runtime logs and performance metrics.

Artifact Setup (incl. Inputs)

Hardware: The experiments were conducted on the Frontier supercomputer, which consists of 9,472 compute nodes, each equipped with 2 x 1.9 TB Samsung PM9A3 M.2 NVMe SSDs and interconnected via the Cray Slingshot network. Although the main experiments were performed on Frontier using 1-1024 nodes with 8 GPUs each, the scalability of the FT-Cache system has also been successfully tested on the KISTI(Korea Institute of Science and Technology Information) Neuron cluster, using 1-4 nodes, demonstrating its flexibility across different system scales. The KISTI Neuron cluster is equipped with NVMe SSDs ranging from 2.9 TB to 3.5 TB, and interconnected via Infiniband HDR 200 Gbps.

Software: The FT-Cache system, based on HVAC, was implemented in C++ and includes the following dependencies:

- Mercury HPC communication library (version 2.0.0)
- GCC (version 9.3.0)
- CMake (version 3.18.4)
- Python (version 3.8.5)
- Conda environment setup is provided in the 'environment.yml' file included in the repository.

To set up the environment, use:

```
conda env create -f environment.yml
conda activate ft-cache-env
```

FT-Cache requires the use of 'LD_PRELOAD', as it runs with the DL application in a fault-tolerant environment. For the most up-to-date setup and execution instructions, please refer to the latest version of the aforementioned GitHub repository.

Datasets / Inputs: The primary dataset used is the CosmoFlow dataset, which has a size of 1.3TB. The dataset must be preloaded onto the PFS before running the experiments. Detailed dataset information, including download instructions, is provided in the README file on the GitHub repository.

Installation and Deployment: To compile and deploy the FT-Cache system, first, allocate the necessary resources using a command like `salloc -A $ACCOUNT -J $USAGE -t $TIME -p batch -N $K -C nvme`. After allocating resources, navigate to the server directory and start the FT-Cache server using `srun -n K -c K ./ftc_server 1 &`, ensuring that the environment variable `FT_CACHE_SERVER_COUNT` is set to 1. Next, set the data directory for FT-Cache using `export FT_CACHE_DATA_DIR=$FT_CACHE_PATH/build/src`. Test the FT-Cache setup by running `LD_PRELOAD=$FT_CACHE_PATH/build/src/libftc_client.so srun -n K -c K ../tests/basic_test`. Finally, you can run your DL application with FT-Cache by using the command `LD_PRELOAD=$FT_CACHE_PATH/build/src/libftc_client.so srun -n K -c K $PATH_TO_YOUR_APP`. Timeout configurations can be adjusted in the source files `ftc_comm_client.cpp` and `ftc_client.cpp` by modifying the `TIMEOUT_SECONDS` and `TIMEOUT_LIMIT` parameters, respectively. Further details and the latest updates can be found by checking the provided GitHub repository.

Simulating Node Failures: To assess the fault tolerance of the FT-Cache system, node failures were simulated during the experiments by randomly disabling nodes after the first epoch of training. This was accomplished using the SLURM job scheduler with the command `sacct update NodeName=node_name State=DRAIN`, which isolates selected nodes at random intervals. This approach ensured that the failure events closely mimicked real-world scenarios, allowing for a robust evaluation of the system's ability to maintain performance and data integrity under unexpected hardware failures. The timing and node selection were randomized to prevent any bias in the results. The simulation code used for these experiments can be found in the `load_distribution_simul.cpp` file within the `tests` directory in the GitHub repository.

Artifact Execution

The experimental workflow consists of two primary setups, corresponding to the two versions of the artifact, which are referred to as the PFS Redirection Approach (A_1) and the Elastic Recaching Approach with Ring Hash (A_2):

1. PFS Redirection Approach (A_1):
   - Task T_1: Data loading and preprocessing on node-local NVMe storage. The dataset is split across the nodes, and each node caches its portion of the data.
   - Task T_2: Execution of the FT-Cache system with simulated node failures. During this task, the system detects failures, and redirects the corresponding requests to the PFS.
   - Task T_3: Post-execution analysis, where runtime performance and load distribution metrics are collected and analyzed to evaluate the effectiveness of the PFS redirection mechanism.
2. Elastic Recaching Approach with Ring Hash (A_2):
   - Task T_1: Similar to the PFS Redirection Approach, but with an enhanced mechanism that uses a hash ring for better load distribution and fault tolerance.
   - Task T_2: Execution of the FT-Cache system with enhanced fault tolerance features, using the hash ring to redistribute data among surviving nodes instead of relying solely on PFS.
   - Task T_3: In-depth analysis focusing on the scalability of the system as the number of nodes increases from 1 to 1024. Metrics collected include runtime performance, recovery time, and load distribution across nodes.
3. The simulational experiment follows a similar workflow but incorporates adjustments for the simulated environment. This includes setting specific simulation parameters such as the number of virtual nodes, epochs, and failure injection methods to replicate the conditions described in the paper. For instance, simulations were run 500 times using 1024 physical nodes, and different numbers of virtual nodes were configured to observe their effect on load distribution after node failure. The simulation outputs can be compared against the results presented in the paper, particularly Figures 3-6.

Artifact Analysis (incl. Outputs)

The artifact produces logs and performance metrics. These outputs can be directly compared to the figures and tables in the paper, specifically Figures 3-6, to validate the results.