EROICA*: Online Performance Troubleshooting for Large-Scale Model Training

Yu Guan[1], Zhiyu Yin[1], Haoyu Chen[1], Sheng Cheng[1], Chaojie Yang[1], Kun Qian[1], Tianyin Xu[2], Pengcheng Zhang[1], Yang Zhang[1], Hanyu Zhao[1], Yong Li[1], Dennis Cai[1], Ennan Zhai[1]
[1]*Alibaba Cloud,* [2]*University of Illinois Urbana-Champaign*

Abstract

Troubleshooting performance problems of large model training (LMT) is immensely challenging, due to unprecedented scales of modern GPU clusters, the complexity of software-hardware interactions, and the data intensity of the training process. Existing troubleshooting approaches designed for traditional distributed systems or datacenter networks fall short and can hardly apply to real-world training systems. In this paper, we present EROICA, the first online troubleshooting system that provides both fine-grained observation based on profiling, and coverage of all machines in GPU clusters, to diagnose performance issues in production, including both hardware and software problems (or the mixture of both). EROICA effectively summarizes runtime behavior patterns of LMT function executions via online profiling, and leverages differential observability to localize the root cause with minimal production impact. EROICA has been deployed as a production service for large-scale GPU clusters of ∼100,000 GPUs for 1.5 years. It has diagnosed a variety of difficult performance issues with 97.5% success.

1 Introduction

Troubleshooting performance problems of Large Model Training (LMT) has become a grand challenge driven by unprecedented increases in AI model sizes and GPU-based training infrastructures. LMT undertakes complex hardware-software interactions; unexpected behavior of hardware components (*e.g.*, GPUs and their interconnects) to software code and configurations (*e.g.*, data loaders, communication, and PyTorch), could lead to performance issues. With rapid scaling of LMT, occurrences of performance problems inevitably increase accordingly, resulting in significant wastage of compute resources and production incidents [35, 39, 40].

In practice, performance troubleshooting of production LMT relies on data from either *online monitoring* or *offline profiling*. Unfortunately, neither is ideal, and engineers often face difficult trade-offs between observation granularity and cluster-wide coverage. Online monitoring must be coarse-grained to cover the entire GPU cluster, including all LMT training processes (*i.e.*, workers) in all machines, without impacting performance. Offline profiling provides fine-grained observation but incurs substantial overhead and generates a huge amount of data that cannot be consumed in real time, so it cannot be enabled in real large-scale training; instead, engineers usually create a smaller test training environment for profiling, which lacks timeliness and is not always possible.

Most online monitors use vendor tools to collect performance counters of hardware components (*e.g.*, GPUs [14], CPU [7], CPU-GPU [6], and NICs [7, 12, 36]) during LMT, or collect timestamps of specific user function executions [24, 25]. To avoid production impacts, typically the hardware sampling is only in second granularity, and only a few important user functions can be selected for monitor. However, in our production experience, the coarse-grained observability can only detect general symptoms (*e.g.*, performance degradation in forward/backward phases or in communication modules), but cannot pinpoint fine-grained root causes (*e.g.*, which line of code or which network link caused the degradation).

Offline profiling (*e.g.*, Torch Profiler [21], Nsight System [19], and Nsight Compute [18]) records all function executions (*e.g.*, Python call stacks, CUDA executions) in LMT and high-frequency hardware sampling (*e.g.*, at 200k Hz). Based on these profiling data, a troubleshooting system can detect subtle anomalies in the program behavior of all LMT workers. But, given the sheer volume of profiling data across the stack, analyzing the data of all workers at runtime cannot be done at scale, so it cannot serve as an online troubleshooting approach for production LMTs.

Contributions. We present EROICA, the first online performance troubleshooting system that offers both fine-grained observability as in offline profiling and cluster-wide coverage as in online monitoring (*e.g.*, can be enabled simultaneously at all LMT workers to cover the entire GPU cluster), effectively localizing root causes of LMT performance issues.

EROICA pinpoints root causes of performance issues in both hardware and software components. It scales to modern AI models on large-scale GPU infrastructure— identifying abnormal function executions for a 3,400-GPU LMT only takes 3 minutes (with the ability to analyze a 1,000,000-GPU LMT in 7 minutes). EROICA supports all training frameworks using PyTorch. To use EROICA, one only needs to import it in the Python source code for LMT without other modifications.

The key insight of EROICA is to develop highly efficient differential observability of LMT profiling data. EROICA does not attempt to directly analyze raw profiling data of all workers (*e.g.*, compare all function events between all workers). Instead, EROICA summarizes the *runtime behavior pattern* of causally related LMT functions on the critical path of the observed performance anomaly. We show that it is possible to concisely summarize each function's runtime behavior patterns using statistical metrics (*e.g.*, execution duration, hard-

*The title EROICA shares its name with Beethoven's Symphony No. 3 (Op. 55), a pinnacle of Classical music and a starting point of Romanticism, as EROICA highly optimizes classic monitoring and profiling for LMT performance troubleshooting while making a first attempt toward future AIOps.

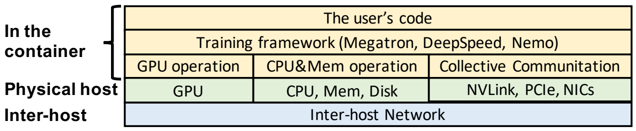


Figure 1: The full stack of large model training.

ware resource utilization, *etc.*), without loss of diagnosability. Since the size of runtime behavior patterns is orders of magnitudes smaller than raw profiling data, the abnormal behavior can be efficiently identified by pattern comparison, leveraging the asymmetric architecture of model training.

We have developed EROICA as a production service in our LMT infrastructure with ∼100,000 GPUs and deployed it for 1.5 years. EROICA has been helping our customers to troubleshoot various LMT performance issues which can hardly be addressed by existing practices or state-of-the-art techniques, including hardware and code-level issues.

Summary. This paper makes the following contributions:

- We present EROICA as the first profiling-based online performance troubleshooting system for LMT in production.
- We describe the technique of summarizing runtime behavior patterns of LMT functions, which enables highly efficient differential observability for LMT.
- We share our production results of EROICA.

2 Background and Motivation

2.1 Performance Issues in LMT

Large model training (LMT) requires the cooperation of multiple hardware and software modules (Figure 1). The hardware typically includes GPUs [15, 16], CPUs, DRAM, PCIe, NVLinks [20], NICs [4, 13], and inter-host networks [34, 39]. Software modules include GPU operators, communication libraries like NCCL [2], training frameworks (*e.g.*, Megatron [1], Nemo [17]), containers, *etc.*. Malfunctions in any of the above components can lead to performance issues, such as throughput decrease, fluctuations, or training blockage. In this paper we do not focus on LMT crashes, which can be explicitly identified by error messages.

Figure 2 breaks down all LMT performance issues in our production GPU cluster with ∼100,000 GPUs over nine months. These issues manifest as slow computation or communication, but with diverse root causes—44.4% of them are caused by hardware issues, and 48.2% are software issues.

Computation slowdowns can be caused by hardware (*e.g.*, GPU throttling), deficient user code, misconfigurations, or incompatible libraries. For example, in our customers' models, some training code frequently transfers data between CPUs and GPUs, introduces excessive synchronization, uses outdated PyTorch versions, poorly manages memory, or contains custom functions that run unexpectedly long.

Communication slowdowns can stem from physical networks, including NVLink, PCIe, NICs, and switches with

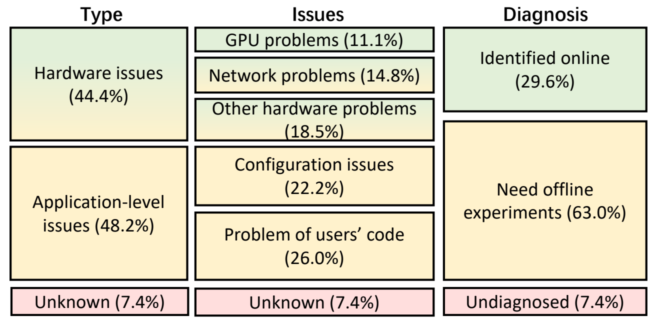


Figure 2: An overview of LMT performance issues.

Table 1: Representative performance diagnosis tools for LMT

Usage	Tool	Diagnostic Information			
		GPU/DRAM/ PCIe/NVLink	NIC	Python Events	Kernels
Online	DCGM [14]	1Hz	✗	✗	✗
	MegaScale [35]	✗	1kHz	✗	✓
	Dynolog* [6]	0.1Hz	0.1kHz	✗	✗
	NCCL Profiler [25]	✗	✗	✗	✓
	eBPF [24]	✗	✗	✗	✓
Offline	Nsight Systems [19]	10-200kHz	1kHz	✗	✓
	Torch Profiler [21]	✗	✗	✓	✓
Online	**EROICA**	10-200kHz	1kHz	✓	✓

*Dynolog supports Torch Profiler as a plugin, to collect Python and kernel information. But, its diagnosis is based only on hardware information and thus is not regarded as a tool based on Python and kernel events.

optical modules. Although major communication throughput reduction is easy to detect, pinpointing the exact machines or hardware components is much more challenging. The difficulty is because most communications in LMTs are collective operations—a slowdown in any network link can proportionally reduce the communication throughput of many workers simultaneously. Besides, bugs in communication libraries, RDMA, and workload managers (*e.g.*, Slurm [22]) can also lead to communication slowdowns.

Furthermore, system services used by LMT can lead to performance issues as well. For example, storage services can affect the performance of data loading or checkpointing during LMT. Management services like load tests and monitoring are co-located on each host; unexpected behavior can lead to resource contention, slowing down the LMT.

2.2 Common Practices

Before EROICA, we relied on existing tools or techniques to troubleshoot performance issues of LMT (Table 1). However, none of them meets our needs in production.

Monitoring. Our GPU clusters widely deployed online monitors on hardware performance counters of GPUs, PCIe, NVLink, and the inter-host network with second-level sample rates, using existing techniques (*e.g.*, in DCGM [14], Dynolog [6], and MegaScale [35]). There are also tools that collect function-level information online. NCCL Profiler [25] (and related work such as Mycroft [31] and Aegis [32]) focuses on collecting collective communication function events, whereas eBPF [24] techniques (*e.g.*, bpftrace [23]) collect events from system calls or other user-specified functions. However, as shown in Figure 2, only 29.6% of performance issues can be diagnosed and root-caused by these tools.

The key reason is that online monitors are coarse-grained and unaware of detailed behavior, creating a fundamental gap between observations and root causes. First, performance issues caused by configurations or code are invisible to hardware monitoring. Second, most warnings from monitors are false positives [32], they do not necessarily indicate performance issues in LMT; they can also be results of temporarily high pressure on hardware (*e.g.*, excessive CNPs) or correctable errors (*e.g.*, XID 63,64, and 92). Third, some hardware misbehaviors are fine-grained and bursty (*e.g.*, <1ms), which hardware monitors at second-level granularity may miss. Finally, even simple issues, which are supposed to be detected by hardware monitors like link down, still have a chance to evade, because in production, hosts are dynamically added or removed, making it challenging to achieve 100% hardware monitoring coverage [32].

In practice, engineers usually analyze the code behavior of LMT in conjunction with hardware throughput and warning information, which have not been automated online.

Profiling. Profilers provide fine-grained observation of LMT and are used as a supplement to online monitors. Nsight System and Torch Profiler record all activities in LMT execution with microsecond-level hardware sampling, which provide sufficient information to identify the root cause of sophisticated performance issues.

However, given their significant overhead (*e.g.*, Torch Profiler generates 100+ MB profiling data for one worker per second), profiling can only be done in a small testbed, or for a few specific workers (*e.g.*, only for rank 0) among thousands of workers in an online LMT. Engineers must first reproduce the observed performance issues in production in the testbed before using a profiler. Unfortunately, reproducing production issues is not always possible or timely. Besides problems that only manifest at a large scale, a practical problem is that user code is often confidential and cannot be disclosed to the GPU cluster providers. Engineers often have to reproduce production issues using similar open-source models, which further reduces the chances of reproduction. Although we invested tremendous efforts to reproduce LMT performance issues reported by our customers, there are still 7.4% of performance issues that cannot be reproduced. For the ones that we eventually reproduced, it takes days to months of effort.

2.3 Goals and Challenges

Our goal is to significantly improve both efficiency and effectiveness of LMT performance troubleshooting in production at scale, with (1) *fine-grained observability:* profiling all function execution events during LMT, with fine-granularity hardware sampling, (2) *efficiency and scalability:* as fine-grained observability leads to large volumes of profiling data, online troubleshooting must be efficient and scalable to large GPU clusters; (3) *low overhead:* introducing no performance impact on the routine LMT training, and minimizing the requirements for computation power, network bandwidth, and storage. We address the following challenges:

Challenge 1: Overhead of fine-grained observability.

- *Profiling generates a huge amount of data.* The size of profiling data of a single worker, including application-level tracing (*e.g.*, the timeline of function execution) and hardware metrics (*e.g.*, using nsys [19] for GPU, NVLink, and PCIe, in 10k Hz), is typically 100 MB per second. Suppose an LMT requires 10,000 GPUs, a profiler can generate $\sim$1 TB data per second, which is untenable.
- *Profiling may reduce LMT performance.* Profiling may reduce LMT training throughput (§6.4). Most customers cannot afford periodic profiling for their LMT jobs.

Challenge 2: Needles in the haystack.

- *Extracting important information.* Fine-grained profiling data far exceeds the volume that can be efficiently analyzed in real time (*e.g.*, 1 TB/s for a 10,000-GPUs LMT). In fact, it is even hard to aggregate all the data; so, algorithms that assume seeing all the data are hardly practical. Dropping or downsampling is not ideal, as it may reduce important details and thus impair the diagnosability.
- *Avoid expensive coordination.* In production, hosts are not always perfectly synchronized in clock time. Even with protocols like Network Time Protocol (NTP) [38], the error could be $\sim$10 milliseconds. Algorithms that compare timestamps of events between workers would not work—a lot of function executions are on the millisecond or even microsecond scale. A decentralized algorithm is needed.

3 Main Idea

The key idea of EROICA is to develop highly efficient differential observability of LMT functions, leveraging homogeneous architectures of LMT. We use the term "*function*" to refer to any procedure in LMT, including Python functions, GPU/CPU kernel functions, memory operations, *etc.*. Compared to traditional systems, LMT function executions exhibit two new characteristics.

- Runtime behaviors of function executions (*e.g.*, the start/end time of execution, and the hardware usage during execution) are expected to be similar across workers or, when not identical (*e.g.* in heterogeneous computation like expert/pipeline parallelism), to follow a relatively stable distribution, as a model usually has high symmetry to be distributed to multiple workers (examples in Appendix E).
- In our production, once a training job encounters a performance issue, it always persists until remediation. This

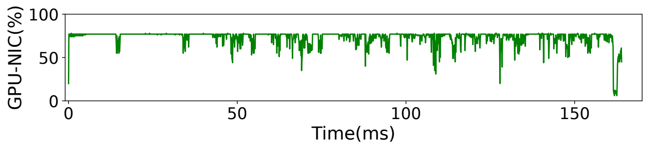


Figure 3: PCIe bandwidth utilization (from GPU to NIC) during a ring communication without network issues.

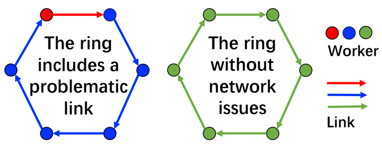


Figure 4: The ring topology with and without a problematic link. We illustrate a link with only six workers for simplicity. There are 32 in our experiment.

is because training iterations repeatedly execute the same functions, and neither hardware faults nor code bugs are self-healing, so repeated execution continues to exhibit the same performance degradation.

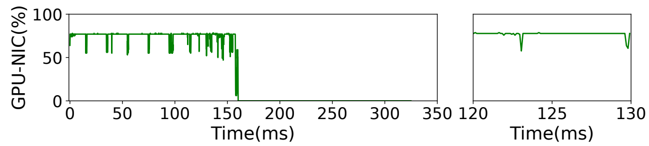
Therefore, most performance issues can be diagnosed by observing abnormal function runtime behaviors in comparison to all other function executions. We elaborate on this idea using a representative example of an LMT performance issue caused by ring communication functions.

Ring communication in LMT (like NCCL [2] AllReduce and AllGather) connects workers in ring-like topologies for direct data exchange with only their adjacent neighbors. A ring topology utilizes both NVLinks (intra-host) and GPU-NIC (inter-host) links. In this example, a NCCL AllReduce group with 32 GPUs is distributed on 4 hosts, each hosting 8 GPUs, with every pair of GPUs sharing two bonded NICs. A malfunctioning NIC downgrades a bond by 50% and creates a performance issue for the NCCL AllReduce function. Since multiple rings typically share the intra-host network (*e.g.*, NVLink) but use different NICs for inter-host communication, we focus on each worker's GPU-NIC throughput during NCCL AllReduce (Figure 3), to distinguish the communication behaviors of different rings and different workers.

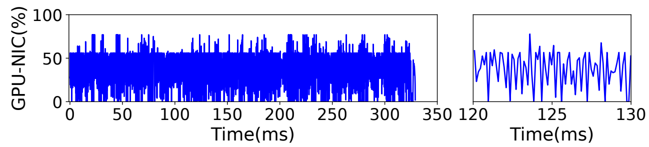
After downgrading a network bond, all workers' GPU-NIC throughputs are categorized into three different patterns.

- For workers not included in a ring using this downgraded bond (green nodes in Figure 4), the throughput pattern (Figure 5a) is identical to the healthy state (Figure 3).
- For workers in a ring using the downgraded bond but not directly connecting to this bond (blue nodes in Fig. 4), the average throughput is lower (Figure 5b) with fluctuations.
- The worker that directly connects to the downgraded bond (red node in Figure 4) also has low average throughput (Figure 5c) but is more stable than Figure 5b.

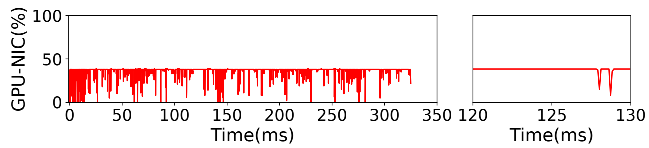
To explain the low average throughput of Figures 5b and 5c compared with Figure 5a: during the ring communication,



(a) GPU-NIC links at maximal throughput.



(b) GPU-NIC links of lower throughput and high fluctuation.



(c) GPU-NIC link of lower and stable throughput (slow link).

Figure 5: GPU-NIC throughput pattern. Left figures are the bandwidth utilizations during the execution of a ring communication function, and right figures show enlarged views of local areas for better observation.

the NCCL communication library constructs multiple rings, each using different NICs, to link all workers head-to-tail. If a ring contains no slow link, all of its links operate at maximal throughput. For rings including a slow link, the average throughput of all workers included in this ring drops.

To explain the fluctuation in Figure 5b: ring communication always transfers data in small chunks one by one. At each stage, each worker transfers a chunk to the next worker, and does some computation based on the chunk it received. With a slow link in a ring, although other links can still reach maximal throughput, they have to wait for the slow link after each stage of chunk transmission. So their throughput fluctuates between zero and the maximal, with a high standard deviation. The slow link does not need to wait, so its throughput remains low with a small standard deviation as shown in Figure 5c.

Insights. The above example demonstrates three important opportunities: (1) performance issues can be observed by profiling the behavior of function executions, and we can use differential observability to localize the offending function executions with abnormal behavior (*e.g.*, low average GPU-NIC throughput without fluctuation); and (2) we do not need to analyze fine-grained raw observability data of all the functions; instead, we only need to summarize their *runtime behavior patterns*. In the ring communication example, each worker only needs to provide two numbers (mean and standard deviation of GPU-NIC throughput) as the pattern of the ring communication function; (3) Fortunately, *runtime behavior patterns* (*e.g.*, mean and standard deviation) can be independent from absolute timestamps, which can be compared across multiple hosts without any clock time synchronization, solv-

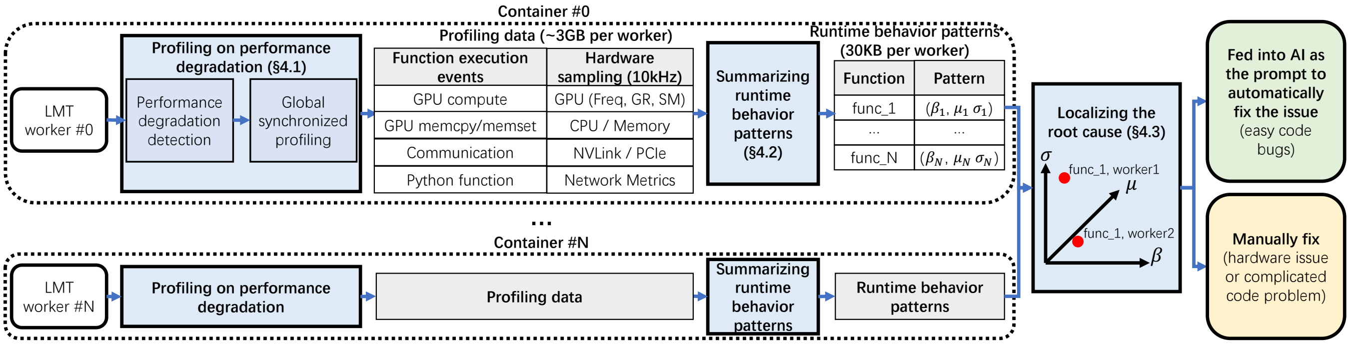


Figure 6: Overview of the EROICA system.

ing the Challenge 2 in §2.3. In summary:

> Most (if not all) LMT performance issues can be diagnosed by highly efficient differential observability on runtime behavior patterns of individual LMT functions.

4 EROICA Design

EROICA is an online performance troubleshooting system for large-scale model training (LMT). It embodies the insights discussed in §3 in a production service for customers who run LMT on our GPU clusters. EROICA employs a highly efficient approach to detect performance issues by online profiling on demand, introducing little overhead on LMTs. When a performance issue is detected, EROICA automatically troubleshoots it. EROICA uses online profiling to collect fine-grained information, but it does not aggregate raw profile data of every function; instead, EROICA summarizes the runtime behavior patterns of each function at every worker node and aggregates only the concise behavior patterns. The diagnosis is performed on the aggregated behavior patterns based on the differential observability principle.

Figure 6 gives an overview of EROICA. EROICA has three main components: (1) detecting performance degradation of LMT to trigger online profiling simultaneously for all workers in LMT, (2) summarizing runtime behavior patterns of each function from raw profiling data, and (3) a centralized localization algorithm that pinpoints the root-cause function based on the behavior patterns. Both (1) and (2) are deployed per worker, while (3) is global.

Usage. To deploy EROICA, GPU cluster providers do not need access to the user code. The user only needs to import EROICA in the LMT code (`import EROICA`); no other code change is needed. Behind the line of import, EROICA initializes the detector of performance degradation, registers handler functions to start profiling, and creates a daemon process for summarizing and uploading behavior patterns. Figure 7 illustrates the output of EROICA. As a function-centric approach, EROICA pinpoints which functions on which workers are executed abnormally, and describes how they behave differently

Abnormal function execution	Function runtime behaviors		
	Duration	Avg Resource util.	Resource util. std
dataloader.py: socket recv on all workers	500ms (480ms↑)	98% CPU freq	1%
Ring Allreduce on worker 7	200ms	37% PCIe Tx (38%↓)	5%
CUDA GEMM on workers {0,1,2,3}	300ms (200ms↑)	33% GPU SM (66%↓)	5%

Figure 7: An example of EROICA output.

from expectation or from other workers. In Figure 7, all workers encounter slow socket receive in data loading, indicating slow storage I/O. Worker 7 is slow in collective communication, indicating a network degradation on its connected link. In addition, worker 0,1,2, and 3 execute `GEMM` (a CUDA kernel function [10]) slowly because of not fully utilizing GPUs, indicating the problem of GPU throttling.

EROICA's output is directly fed into an AI assistant to try automated fixing: for simple code bugs, the assistant may automatically patch the code (a real case in §6.3), whereas hardware faults or complex code issues typically still require human effort to fix (*e.g.*, replacing the problematic hardware).

4.1 Profiling on Performance Degradation

Profiling an LMT task online introduces overhead. To minimize the overhead while collecting profiling data timely when performance issues occur, EROICA automatically detects performance degradation and then performs profiling simultaneously for all workers, ensuring the problematic behavior is included in the profiling data.

For an LMT task, the time spent on one training iteration is the key performance metric. EROICA devises a solution to monitor the training time of each iteration without accessing user code, and then triggers online profiling if the iteration time is abnormal. Note that approaches that require accessing user data (such as logs) do not work with our usage model.

Indicators of iteration time. A PyTorch training iteration always involves several `dataloader.next()` calls, followed by several `optimizer.step()` calls (the number depends on training parameters). In this sequence of events, the duration from the first `dataloader.next()` to the last `optimizer.step()` is regarded as the duration of a complete training iteration.

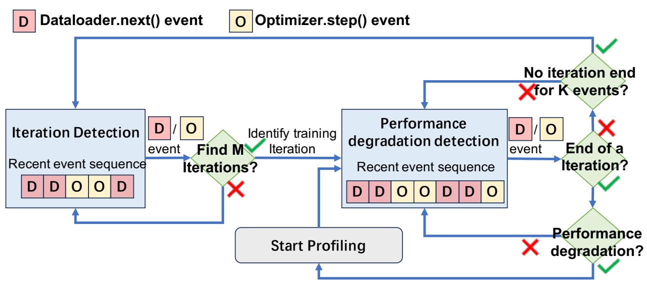


Figure 8: EROICA's performance degradation detection.

When it is `import`-ed, EROICA wraps the two functions by adding time counters. Basically, EROICA replaces the two PyTorch functions with the wrapped versions. Since the two functions are Python functions (which are not compiled before LMT executions), the replacement is done at runtime.

Iteration detection. During an LMT, EROICA continuously records event sequences of `dataloader.next()` and `optimizer.step()`. After detecting M (=10 in practice) identical sequences starting with `dataloader.next()` and ending with `optimizer.step()`, this sequence is defined as the *training iteration sequence*.

Performance degradation detection. EROICA continuously monitors incoming events of `dataloader.next()` and `optimizer.step()` with the identified training iteration sequence. Each time EROICA successfully matches a complete training iteration sequence, it records the duration of that iteration. EROICA considers that performance degradation has occurred in the following two situations: (1) The average duration of the recent N (=50 in practice) iterations exceeds the recent shortest iteration time by more than 5%. (2) The current training iteration sequence has not yet been fully matched, but the time elapsed since the last event received is at least 5$\times$ the average iteration duration (indicating the training is blocked). If EROICA fails to match a training iteration after K (=200 in practice) consecutive event receptions, it goes back to the previous iteration detection phase to redetect the training iteration sequence. While this does not often happen in common LMT programs, it significantly enhances the robustness of EROICA when users implement special functionalities in their code, to make sure the algorithm always works. Figure 8 illustrates the procedure.

Global synchronized profiling. Each LMT worker connects to an EROICA daemon. When EROICA detects a slowdown, it notifies all daemons via TCP; each daemon signals its worker to invoke a pre-registered handler, ensuring profiling runs in the LMT main thread (required by some APIs, *e.g.*, CUPTI in Torch Profiler).

EROICA ensures synchronized profiling across workers via iteration ID: rank-0 continuously report the current iteration ID; upon a profiling trigger, the rank-0 daemon computes unified start/stop iteration IDs for profiling (with the start set a few steps ahead to ensure no worker would miss it), and other daemons periodically poll these IDs and start/stop

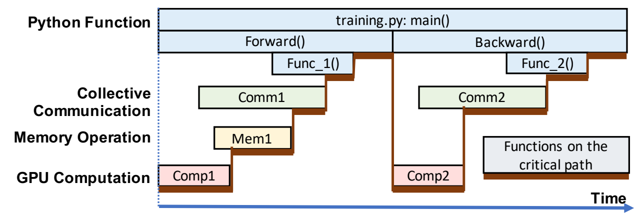


Figure 9: The critical path of LMT performance.

profiling accordingly.

A profiling session lasts for 20 seconds by default (configurable). We use Torch Profiler to collect execution events (Python/CPU ops, memory ops, and CUDA kernels), and `nsys` to sample hardware metrics at 10 kHz (GPU, DRAM, NVLink, PCIe, and network). Each daemon aggregates and collects the profiling data.

4.2 Summarizing Behavior Patterns

EROICA summarizes behavior patterns of each function at the worker level. Note that EROICA only concerns functions on the critical path of LMT (Figure 9)—only these functions contribute to the end-to-end performance, despite the huge number of function executions in an LMT.

EROICA's solution is (1) finding the function execution events on the critical path, including GPU computation kernels, collective communication functions, memory operations, Python functions, and all other functions executed in LMT; and (2) clustering all execution events of each function (for Python functions, the entire call stack must be identical to be considered the same function), then defining several patterns to summarize the behavior of each function.

Critical path. As shown in Figure 9, LMT assigns different types of function with priorities (higher is more critical): GPU compute kernels $>$ memory operations (*e.g.*, malloc, memcpy) $>$ collective communication kernels (*e.g.*, NCCL AllReduce) $>$ Python functions. A function's execution (or a subinterval of it) is on the LMT's critical path iff no higher-priority function is executing during that time. For Python functions, we further require that they run in the training thread (*e.g.*, not spawned by `_bootstrap`) and have no executing child calls.

The rationale for such a definition is that a well-optimized LMT should try to keep GPUs busy, so in general we should focus on GPU kernel executions and function executions during GPU idle time. Therefore, all function executions are prioritized based on their relevance to GPU computation. Only when a function unrelated to GPU computation fails to overlap with GPU computation (indicating it blocks GPU computation), it should be regarded as a possible problematic function. Reducing its execution time can improve end-to-end training performance. Otherwise, it remains irrelevant to performance since it does not block GPU computation.

Note that one could craft a counterexample against EROICA, for example, by fully overlapping communication with GPU computation, making communication stalls invisible on the critical path. However, in production LMT, computation and various communications have strict dependencies. While some communication can overlap with computation, we have never seen real workloads where all communication is fully overlapped; thus it still partially appears on the critical path and can be detected by EROICA.

Runtime behavior patterns. For each function f running on worker w, we define its pattern of behavior $P_{f,w}$ as a 3-dimensional vector:

$$P_{f,w} = (\beta_{f,w}, \mu_{f,w}, \sigma_{f,w}) \tag{1}$$

$\beta_{f,w}$ represents the percentage of the profiling window that function f spends time on the critical path of worker w.

$$\beta_{f,w} = \frac{\int_{t \in T} C(f,w,t)}{|T|} \tag{2}$$

where T is the duration of the profiling window (*e.g.*, 20 seconds in §4.1), and

$$C(f,w,t) = \begin{cases} 1 & f \text{ is on } w\text{'s critical path at time } t, \\ 0 & \text{Otherwise .} \end{cases} \tag{3}$$

Base on the definition, we have $\beta_{f,w} \in [0,1]$.

$\mu_{f,w}$ represents the average hardware resource utilization during all executions of function f on worker w. In this context, "resource" refers to hardware resources that determine performance of function f. For example, GPU computation kernels correspond to GPU SM frequency, Python functions to CPU utilization, intra-worker collective communication to NVLink utilization, and inter-worker collective communication corresponds to the bandwidth utilization between GPU and NIC, due to the significantly lower NIC bandwidth compared to intra-worker NVLink bandwidth.

$$\mu_{f,w} = \frac{\sum_{e \in E_{f,w}} |L(e)| avg_U(L(e))}{\sum_{e \in E_f} |L(e)|} \tag{4}$$

where $E_{f,w}$ represents all execution events of function f on worker w in the time window, so e represents each one of them. $L(e)$ represents the *critical execution duration* of e. $avg_U(L(e)) \in [0,1]$ represents the average resource utilization during the critical execution duration. We have $\mu_{f,w} \in [0,1]$.

The definition of critical execution duration, $L(e) = [l_c, r_c]$, is not trivial, and is a subinterval of f's function execution duration $[l, r]$, as shown in Figure 10. This is because in practical LMT tasks, particularly within collective communication functions, numerous synchronization operations are involved. Workers that enter the collective communication kernel earlier usually perform a part of communication and then wait for others. Consequently, the throughput of network transmission are not continuous. The average resource utilization over the entire function execution does not accurately reflect the real communication performance. Thus, $L(e)$ is defined as the longest interval with intense resource usage among the whole function execution duration. We apply Algorithm 1 to identify L_e with discrete sampling of hardware resource usage.

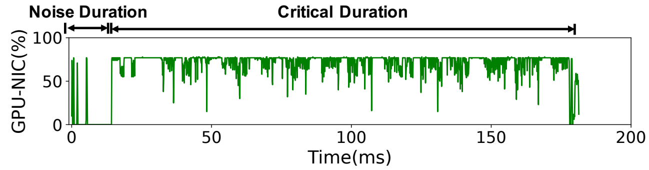


Figure 10: Critical duration of a collective communication function. A worker enters the communication early often need to wait for others, leaving several empty intervals of resource usage (noise). We only consider the critical duration in $\mu_{f,w}$.

Algorithm 1 Finding the critical execution duration

Require: Resource utilization samples $U = [U_1, U_2, \ldots, U_n]$ of function f's execution duration $[l, r]$, where $U_i \in [0,1]$ for $\forall i$

Ensure: Critical execution duration $[l_c, r_c]$

Total resource utilization $S = \sum_{i=1}^{n} U_i$

Initialize search bounds: $g_{\text{left}} \leftarrow 0$, $g_{\text{right}} \leftarrow n$

while $g_{\text{left}} \leq g_{\text{right}}$ **do**

  $g \leftarrow$ midpoint of $[g_{\text{left}}, g_{\text{right}}]$

  if there exists a subinterval $[l', r']$ satisfying:

    1. $\sum_{i=l'}^{r'} U_i \geq 0.8 \times S$

    2. No more than g consecutive zeros in $U_{l'}, \ldots, U_{r'}$

    Update $g_{\max} \leftarrow g$, $[l_c, r_c] \leftarrow [l', r']$

    $g_{\text{right}} \leftarrow g - 1$

  else

    $g_{\text{left}} \leftarrow g + 1$

end while

return Subinterval $[l_c, r_c]$

$\sigma_{f,w}$ represents the standard deviation of resource usage during all function executions of function f on worker w, and its definition is similar to $\mu_{f,w}$:

$$\sigma_{f,w} = \frac{\sum_{e \in E_{f,w}} |L(e)| std_U(L(e))}{\sum_{e \in E_f} |L(e)|} \tag{5}$$

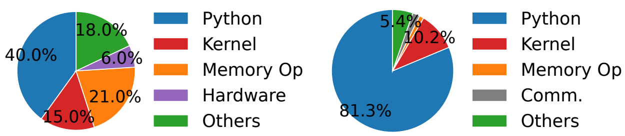
and we also have $\sigma_{f,w} \in [0,1]$.

Note that all three dimensions of $P_{f,w}$ are independent of absolute timestamps. For example, to compute $\beta_{f,w}$, we only need the time difference between the start and end of function execution. This design is the key to performing inter-host pattern comparison without the need for clock synchronization.

Data size of patterns. Figure 11b shows the size of runtime behavior patterns (30 KB) is $10^5\times$ smaller than the raw profiling data ($\sim$3 GB). Patterns of Python functions contributes to the majority of the data volume, because EROICA records the full call stack of each function, which is long (sometimes the call stack of a Python function includes 1,000 letters).

4.3 Localization

EROICA localizes the abnormal function behavior based on the runtime behavior patterns in §4.2. In most cases, the abnormal function behavior can directly pinpoint a single plausible



(a) Raw profiling data (3GB) (b) Runtime behavior patterns (30KB)

Figure 11: Data size of a single worker's runtime behavior patterns compared with its raw profiling data.

root cause. For more complex cases, our engineers make the final call, optionally with the help of an AI assistant.

In LMT execution, all performance issues can be categorized into two types: (1) a common problem of all LMT workers, like hardware misconfigurations and low-efficiency code implementation, or (2) a special problem on only a part of workers, like hardware issues. Correspondingly, given function f executed on worker w, EROICA defines two types of distances, (1) *Distance from expectation* and (2) *Differential distance*, to measure its abnormality.

Distance from expectation. The first is the distance from expectation, $D_{f,w}$. In a well-optimized LMT, we have an expected range for the runtime behavior patterns of most functions (*e.g.*, a Python function f are expected to have a close-to-zero value of β_f because LMT should not be bottlenecked by CPU). Any function with a runtime behavior pattern far from this range can be considered abnormal. When there are a lot of workers with large distance from expectation, it indicates common issues in the LMT, such as low hardware configurations across the cluster or deficient implementations of certain functions deployed on all workers.

To define the distance from expectation, we first need an expected range of patterns for function f, R_f:

$$R_f = [\beta^l_f, \beta^r_f] \times [\mu^l_f, \mu^r_f] \times [\sigma^l_f, \sigma^r_f] \tag{6}$$

R_f is assigned based on our production experience. For example, if f is a Python function, we set $R_f = [0, 0.01] \times [0,1] \times [0,1]$, since customers typically regard performance fluctuations within 1% as measurement noise; deviations beyond this threshold are viewed as substantial regressions and thus trigger diagnostic requests. If f is a collective communication function, we set $R_f = [0, 0.3] \times [0,1] \times [0,1]$. For GPU computation kernel functions, we set $R_f = [0,1] \times [0,1] \times [0,1]$, *i.e.*, we expect a LMT to spend most of its time on GPU computation to fully utilize the GPUs.

Then we can define $D_{f,w}$ as the minimal Manhattan distance from $P_{f,w}$ to R_f:

$$D_{f,w} = \min_{p \in R_f} d_{\text{Manhattan}}(P_{f,w}, p) \tag{7}$$

Differential distance. The second is the distance to other workers, $\Delta_{f,w}$. In a large-scale LMT, the behaviors of different workers should be highly consistent. If some workers exhibit significantly different function behaviors compared to the same functions executed on other workers, it often suggests that these workers are performing special operations that slow down overall training throughput (*e.g.*, different in β-dimension), or that the hardware used by these workers for the function has performance issues that need to be addressed (*e.g.*, different in μ or σ-dmension).

To make comparison between workers, first we make a max normalization to $P_{f,w}$:

$$\hat{P_{f,w}} = \left(\frac{\beta_{f,w}}{\max_{w \in W} \beta_{f,w}}, \frac{\mu_{f,w}}{\max_{w \in W} \mu_{f,w}}, \frac{\sigma_{f,w}}{\max_{w \in W} \sigma_{f,w}}\right) \tag{8}$$

Then we define $\Delta_{f,w}$ to indicate how many workers have different patterns of f's execution from worker w:

$$\Delta_{f,w} = \frac{\sum_{w' \in W_N} I(\hat{P_{f,w}}, \hat{P_{f,w'}})}{N} \tag{9}$$

where W_N is a subset of N workers randomly sampled from the set of all workers W. In practical we set $N = \min(100, |W|)$. I is a function indicating whether the distance between two vectors are below δ:

$$I(x,y) = \begin{cases} 0 & \text{if } d_{\text{Manhattan}}(x,y) < \delta, \\ 1 & \text{otherwise.} \end{cases} \tag{10}$$

In practical, we set δ=0.4 based on our experience.

Note that our definition of $\Delta_{f,w}$ is not based on how far a worker is from other workers (*e.g.*, simply average the distance to other workers), but how unique a worker's behavior is. This is because the pattern $P_{f,w}$ is in three dimensions, and each dimension has different physical meaning. A larger Manhattan distance does not mean a worker's behavior is more abnormal (§3 shows a classical example).

Localizing abnormal function executions. With the two distances, a function f running on worker w is abnormal if:

$$\beta_{f,w} > 0.01 \land (D_{f,w} > 0 \lor \Delta_{f,w} > M_f + kMAD_f) \tag{11}$$

where $M_f = median\{\Delta_{f,w'} | w' \in W\}$ is the median value of distance to peers, and $MAD_f = median\{|\Delta_{f,w'} - M_f| \; | w' \in W\}$ is Median Absolute Deviation [11], a robust measure of statistical dispersion. k is set to 5.

$\beta_{f,w}$>0.01 means the function execution contributes at least 1% to end-to-end performance. For most LMT tasks, no more than 20 functions satisfy this requirement. $D_{f,w} > 0$ indicates the runtime behavior of function f is unexpected based on our production experience, and $\Delta_{f,w} > M_f + kMAD_f$ indicates the value of $\Delta_{f,w}$ is significantly larger than most other workers.

As shown in §4.2, the pattern of a worker is only ~30 KB, even for an LMT with 10,000 workers, the localization algorithm only consumes 300 MB data to output the final result, which is highly efficient to be executed on a single CPU core.

Alternatives. To localize abnormal function executions, we also tried existing clustering algorithms, including DBSCAN [33], HDBSCAN [37], Gaussian Mixture Model [29] and Mean shift [30]. They do not work well in production because they have at least one of the following limitations: (1) failing to distinguish noises and outliers, which is important in our scenario, or (2) having too many hyper-parameters, reducing their general applicability in practical cases.

5 Implementation

We built EROICA with 7K lines of Python code, and deployed EROICA as a service for our production GPU clusters. We address the following practical challenges.

Optimizations of profiling data generation. EROICA relies on Torch profiler to generate a part of raw profiling data (§4). However, Torch profiler introduces overhead to LMT: (1) When the profiling window ends, it requires a long time to prepare and dump the data, and (2) after profiling, it leaves a lot of resources (*e.g.*, hooks for CUDA functions added by CUPTI [5]) in the LMT. They may reduce the performance of CUDA kernel execution even after profiling is finished.

EROICA addresses the problems by optimizing the Torch profiler. First, the Torch profiler always transfers profiling data to Chrome tracing format, and then dumps the data using Kineto APIs [8]. Since Kineto supports dumping data with the same format, we remove the redundant and slow format transformation performed by Torch profiler, and directly apply Kineto to dump data. This reduces the time spent on data generation by 33%. Second, EROICA calls `cuptiFinalize()` to clear all remaining resources in LMT after profiling.

Accessing hardware information within the user's container. In production GPU clusters, an LMT typically runs inside containers; so does EROICA daemons (§4.1). However, GPU cluster providers usually restrict the permissions of user containers—EROICA cannot directly access hardware information. EROICA utilizes Kubernetes [9]'s native "emptyDir" feature to share directories between containers, performing high-frequency hardware sampling through a privileged management container to place data in the shared directory. Therefore, EROICA can obtain hardware profiling data within user containers without loosening permissions of user containers.

Cooperating with other hardware monitors. In most production GPU clusters, coarse-grained hardware monitors are deployed on each physical host, using APIs such as DCGM and PCM to collect hardware metrics in real-time for health checks. However, some information (*e.g.*, GPU metrics) can only be subscribed by one process at a time. EROICA's data generator cooperates with the host's monitoring system via signals (through a shared directory). As EROICA's profiling typically lasts about 20 seconds each time, we do not encounter any conflicts caused by the cooperation.

6 Evaluation in Production

EROICA has been deployed in our production GPU clusters with ~100,000 GPUs for 1.5 years. During this period, there were in total 80 serious LMT performance issues for which we failed to localize the root causes using state-of-the-art techniques described in §2.2 (see Table 2). EROICA successfully identified root causes of 78 of them (success ratio=97.5%), improving the training throughput of these LMTs (the largest one includes 6,144 GPUs) by ~20% to ~100%.

Table 2: 80 serious performance issues identified by EROICA (ONLY includes those not identified by our existing systems)

Category	Root cause	Number of LMT cases
Hardware issues	GPU	2
	CPU	2
	Network	6
Misconfigurations	PyTorch	4
	Communication	6
	Dataloader	5
Low-efficiency code of users		45

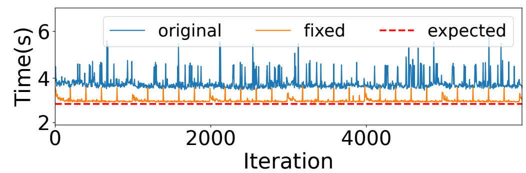


Figure 12: Iteration time of the LMT in Case Study 1.

We present how we used EROICA to diagnose three LMTs with 3,072, 3,400, and 128 GPUs, respectively, to show:

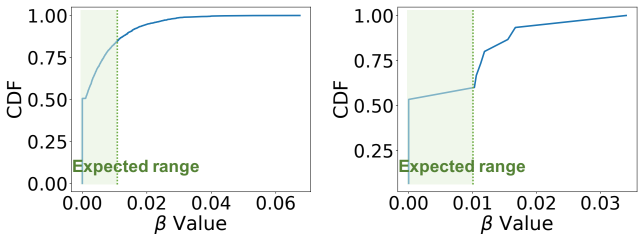
- EROICA can identify real-world performance issues of mixture of multiple hardware and code problems. (§6.1-6.2)
- EROICA's output can be directly fed into AI as the prompt to help customers automatically fix the code bug. (§6.3)
- EROICA makes diagnosis in high efficiency, and does not introduce overhead to the routine training of LMT. (§6.4)

In addition, we present two more case studies in Appendices A and B, including one of the only two (out of 80) cases where EROICA failed to diagnose the issue. In Appendix C, we systematically compare EROICA with state-of-the-arts.

6.1 Case Study 1: Code-level Issues

A text-to-video LMT job on 3,072 Nvidia H800 GPUs is expected to take 3.5 seconds per iteration, but is training at 5 seconds per iteration (see the "original" line in Figure 12).

Problem 1: Low socket throughput in data loader. EROICA outputs abnormal function behaviors on the built-in function `recv_into` of the socket object (a low-level function call in the data loader) on multiple workers. As a GPU-independent function, we expect that the LMT is bottlenecked by it in no more than 1% of the time ($\beta \leq 0.01$, as explained by "expected range" in §4.3). However, as shown in Figure 13a, there are a lot of workers with a high value of β for the `recv_into` function (so they have a high value of $D_{f,w}$). This indicates low efficiency in data loading from the storage.



(a) `recv_into` of 3,072 workers.

(b) `forward` of 16 workers. The data of the rest 3,056 are destroyed by the cluster.

Figure 13: CDF of β values in two functions in Case Study 1.

Problem 2: Inefficient implementation of Python function. Similar to the first problem, EROICA outputs abnormal behaviors on the Python function `forward` with large β values, shown in Figure 13b. This function performs CPU computation and launches GPU kernel functions. A large β value indicates the function is bottlenecked by CPU computation, which needs to be optimized.

Problem 3: Asynchronous garbage collection. Apart from the above functions, EROICA also reports large β values for some other Python functions like `gradmode.py:__init__()` and `_flat_param.py: _get_unflat_views_unaligned`. However, these functions are not CPU-intensive, and their executions with large β are randomly distributed among all workers. This indicates these workers are executing Python garbage collection asynchronously. Each time a worker is collecting garbage, other workers have to wait for it, leading to GPUs being idle and reduced training throughput.

Limitations of existing approaches. The customer did use Torch Profiler and Nsight System to profile their LMT job offline. As discussed in §2.1, profilers generate too much data; the common practice is profiling a few training iterations only in a fixed host (*e.g.*, the rank-0 host). However, none of the three problems were detected, because in most cases the root cause of a low-performance user-level function is due to temporal pause on GPU/host memory allocation (*e.g.*, garbage collection, GPU memory fragmentation, *etc.*) or access to remote storage. This only happens in a few random workers in each iteration, not all workers. Consequently, the offending function is difficult to detect without profiling all workers. In contrast, EROICA has a global view of function behaviors of *all* workers to identify abnormal ones with high efficiency.

Fixes. For the data loading problem, we helped the customer load the input data from our parallel file system rather than the legacy object storage service, which significantly accelerated data loading. For the Python garbage collection problem, the solution was to call the garbage collection every 200 iterations. This guaranteed all workers collect garbage at the same time, thus avoiding mutual waiting. Implementation optimization of the function `forward` is not trivial since it contains many CPU computation operations. At present, the iteration time of the LMT has reached ∼3.6 seconds, as shown in Figure 12.

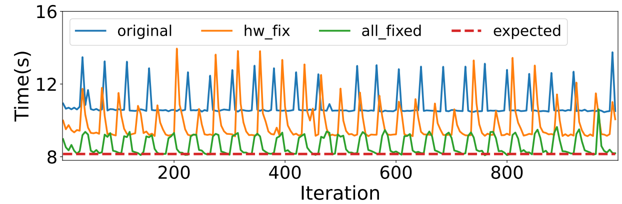


Figure 14: Iteration time of the LMT in Case Study 2.

6.2 Case Study 2: Mixed Code-Hardware Issues

A video generation LMT job on 3,400 Nvidia H800 GPUs is expected to take 8.5 seconds per iteration, but is training at 10.5 seconds per iteration ("expected" and "original" in Fig. 14). And the job crashes every few hours.

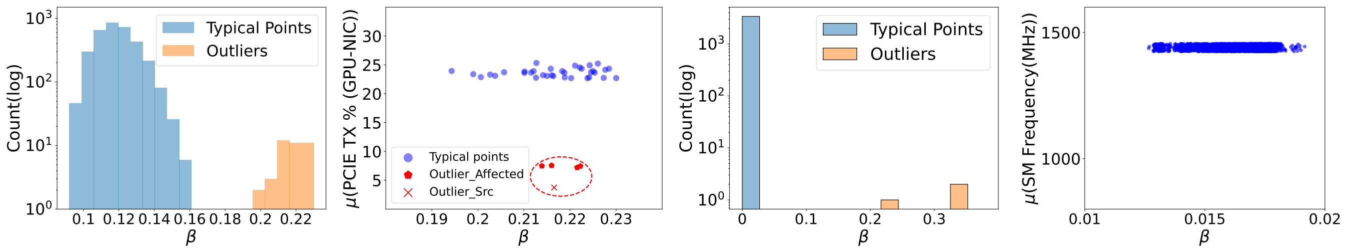
Problem 1: Low cluster network throughput. As shown in Fig. 15a, the β value of most `SendRecv` functions ranges between 9% and 16%. However, our customer believes that the data amount in `SendRecv` is equal across all workers, so the β values of `SendRecv` for all workers are expected to be identical. Moreover, based on the data amount in `SendRecv` and the NIC hardware, the β value is expected to be ∼6%—indicating much lower cluster network throughput than expected. The root cause is that affinity-based flow scheduling is not deployed on this cluster, so inter-host data flow is not optimized.

Problem 2: NIC down. Fig. 15a shows another problem that the β values of `SendRecv` of 40 workers (20%-23%) are much higher than most workers ($9\% - 16\%$). Among the 40, one has high $\Delta_{f,w}$ due to the significantly lower μ than the other 39 (Fig. 15b), indicating a lower bandwidth between the GPU and the NIC. Our network engineers confirmed that the NIC corresponding to the worker is down.

Problem 3: Pin memory in ultra-high frequency. Fig. 15c shows that three random workers spent at most one third of time on `pin_memory` ($23\% - 33\%$ β value). Although this does not happen on the other 3,397 workers, they have to wait for the three workers to finish one iteration together, leading to reduced training throughput. Our customer confirmed that `pin_memory` operations come from `data_loader` processes.

Problem 4: Load imbalance. EROICA finds that almost all GPU computation kernel functions have similar μ but significantly different β across workers. Fig. 15d shows one GPU kernel function for example. The busiest GPU spends 46% more time computing than the most idle one. This indicates that although workers execute GPU kernels with equal performance, some workers launch much more GPU kernels than others, leading to load imbalance. Our customer confirmed that the model is a video-to-video model, for each training iteration, video inputs for workers have different duration, requiring different amount of GPU kernel launches. Workers with less GPU computation have to wait for others with more GPU computation to finish one iteration together; this leads to poor training efficiency (*e.g.*, low Model FLOPs Utilization).

Limitations of existing approaches. For Problem 1, hardware monitoring on the cluster network showed no warning.



(a) The β value of `SendRecv` of 3,400 workers. This value is widely distributed across the 9%-16% range, with a small number (40 in total) of outliers distributed in 20%-23%.

(b) `SendRecv`'s β and μ (GPU-NIC) of 40 workers with high β (Outliers in Fig. 15a). Among the 40 workers, five have lower μ and one of them is significant lower which is the location of NIC down.

(c) The β value of `pin_memory` of 3,400 workers. Most workers (3,397) is close to zero but three of them in range $23\% - 33\%$.

(d) The β and μ value of `chunk_cat_cuda_kernel<float, c10::BFloat16>` of 3,400 workers. All workers have similar μ but different β, indicating the workload imbalance.

Figure 15: Function execution behaviors in the Case Study 2.

The problem was low-efficiency flow scheduling that failed to fully utilize the network. For Problem 2, the affected host was newly added to the cluster, and the monitoring agent had not been updated, therefore, failed to alarm the problem. For Problem 3, the pure code-level problem, it only happened on three of 3,400 workers in one iteration, so it was hard to be captured offline (as explained in §6.1). For Problem 4, although a GPU may receive a long video in one iteration to reach high utilization, due to the randomness of input scheduling, it does not happen in all iterations. Based on our GPU utilization monitor, in general all GPUs have similar utilization in a period of time, so it failed to detect the problem.

Fixes. We removed 20 hosts with the least network throughput in Fig. 15a to mitigate Problem 1, including the host with NIC down (Problem 2), then the iteration time reduced from 10.5s to 9.5s ("hw_fix" line in Fig. 14). For Problem 3, our customer decreased the number of `data_loader` processes to reduce the memory overload. It not only decreased the iteration time (from 9.5s to 9.2s), but also solved the crash problem. For Problem 4, we helped the customer balance the input between workers. Currently, this LMT is training at 8.5 seconds per iteration (the "all_fixed" line in Fig. 14) without crashes. With all of the above efforts, the input data processing rate was improved from 7,173 samples/day to 9,644 samples/day, a 34% end-to-end performance improvement.

6.3 Case Study 3: Diagnose and auto-fix customer-code bugs with AI support

A robotics (embodied AI) model's training of 128 GPUs got stuck for a long time, and the customer requested a diagnosis.

Problem detected: Stucked in dataset preloading. In this stalled training job, EROICA found a single worker behaving differently from the rest. On this worker, a data-loading/preprocessing thread is blocked in `queue.put()` inside `dynamic_robot_dataset._preload()` (waiting on a Python thread lock), indicating the input pipeline is stuck/back-pressured. Meanwhile, other workers are either sleeping in dataset management routines (*e.g.*, `_monitor_config`, `_run_threads`) or waiting in JAX [28] execution, suggesting they are idle and effectively waiting for this worker to make progress. Overall, the hang is most consistent with a Python-side data pipeline deadlock or queue blockage in the dataset prefetch/preload logic.

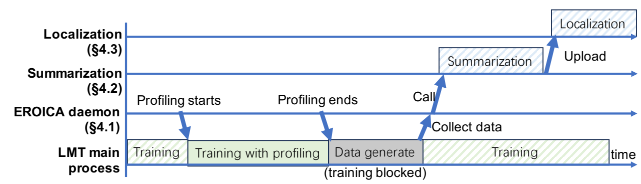


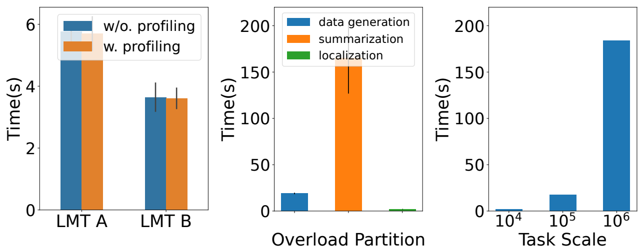
Figure 16: EROICA's overhead in timeline view.

Automatically fixed by AI. We instructed the customer to use EROICA's output (the stuck Python function) together with the relevant code as a prompt to Cursor [27] (Sonnet 4.5), an AI coding tool. Cursor immediately identified the bug: during data fetching, a logging/debug print accessed array[0] on a sharded distributed array, implicitly triggering an `all-gather`. Since this collective was invoked at an unexpected point (not executed by all ranks), it led to a distributed deadlock and stalled the training job. Cursor then automatically generated the fix for the buggy code, and the training job resumed normally.

6.4 System Overhead

Figure 16 illustrates EROICA's overhead. When EROICA starts profiling, the LMT is executed with high-precision profiling for a 20-second window (§4.1). When the profiling window ends, LMT will be blocked by data generation for a short period. After that, LMT continues running with no extra overhead, as the behavior pattern summarization is done in a separate process (using a different CPU core), and the root cause localization is performed remotely from the worker.

Overhead on production LMTs. We evaluate two LMTs: LMT-A of 3,072 GPUs in §6.1 and LMT-B of 3,400 GPUs in §6.2. Figure 17a shows the average iteration time during training, with and without profiling. Figure 17b shows the duration of data generation, pattern summarization, and lo-



(a) LMT iteration time with and without profiling.

(b) Time spent on different components.

(c) Time spent on localization under larger LMT scales.

Figure 17: The overhead introduced by EROICA.

calization. The results show that profiling does not affect the performance of production LMTs; summarization and localization complete within three minutes. The main overhead on LMT is data generation (∼20) seconds, which is incurred just once and accepted by all our customers. Notably, pattern summarization, uploading, and root cause localization occur outside the training process. Although these steps may last for minutes, they introduce zero overhead to the training task. Regarding memory usage, although profiling consumes tens of GBs on each host, this is negligible given that production training hosts typically have 1–2 TB of host memory.

We also evaluate the overhead with different models and configurations in Appendix D to show EROICA is able to make diagnosis with high efficiency in diverse situations.

Scalability to 1,000,000-GPUs LMTs. In EROICA, profiling, data generation and pattern summarization are all distributed in each worker's container and executed in parallel, so their overhead is independent of the LMT scale. The only overhead proportional to the LMT scale is root-cause localization, which is executed on a single CPU core. We generate simulated runtime behavior patterns as input, to evaluate the overhead of localization for larger-scale LMTs. Figure 17c shows that the localization requires only three minutes for a 1,000,000-GPU LMT. Together with the time spent on profiling, data generation, and pattern summarization, EROICA can make an end-to-end performance analysis for a 1,000,000-GPU LMT within 7 minutes.

7 Discussions: Enhancing EROICA with AIOps

EROICA does not guarantee automatic identification of the root causes for all possible performance issues. (1) For most performance issues, EROICA's output of the problematic functions and their corresponding runtime behavior patterns is sufficient to identify the root cause or can be fed into AI to determine the root cause (like the case in §6.3). However, some user-defined Python functions and CPU kernels have complex code and logic. Therefore, even when EROICA identifies that these functions have problems (*e.g.*, excessively long execution times), analyzing the root causes may still require manual inspection of their code. (2) EROICA does not ensure detecting problems outside the training task. For example, a background process running on the physical host may incidentally consume too many resources, leading to a performance issue for the training task in the container on that host. A case study is presented in Appendix B.

To bridge the last-mile gap between abnormal function behavior and the root cause, we are instructing our customers to combine EROICA's output with additional information (the code of the function with abnormal behavior, background processes with system call activities, and hardware configuration/utilization) and generate a standardized prompt for an AI model to diagnose the root cause. In production, while the AI provides correct diagnoses only in a subset of cases, it yields useful hints in most cases, which is particularly valuable for performance issues caused by user code. Therefore, in our research on AIOps (Artificial Intelligence for IT operations) for LMT, EROICA's output serves as the core component for prompt construction.

8 Related Work

Hardware monitoring tools. Accelerators expose rich telemetry through vendor tools. For NVIDIA GPUs, NVML [3] provides a C-based API to query and sample device status (e.g., utilization, temperature, power, memory, and error states). DCGM [14] targets large clusters by offering centralized, scalable GPU health/performance monitoring and diagnostics for fleet management. CPU performance and energy metrics can be collected via Intel PCM [7] (e.g., cache and memory bandwidth statistics). For NICs (e.g., Mellanox), `mstflint` [12] provides access to device counters such as packet/byte throughput and link status.

LMT profiling and analysis. Nsight Systems and Nsight Compute provide complementary profiling for CUDA workloads. Nsight Systems captures an end-to-end timeline across CPU threads, GPU work, and memory transfers, whereas Nsight Compute focuses on per-kernel metrics (e.g., occupancy and memory throughput). Torch Profiler, integrated into PyTorch, reports fine-grained per-operator/per-function events across Python, CPU, and CUDA to attribute training time and identify bottlenecks.

9 Conclusion

This paper presents EROICA, an online performance troubleshooting system for LMT. EROICA presents the process of profiling on performance degradation, behavior pattern summarization, and performance issue localization, which utilize runtime behavior patterns of functions to make diagnosis. We deploy EROICA in real-world production GPU clusters and solve sophisticated performance issues for real users.

Acknowledgements

We thank our shepherd Changhoon Kim, and the anonymous reviewers for their insightful comments. Ennan Zhai is the corresponding author.

References

[1] Megatron-lm: Training multi-billion parameter language models using model parallelism. https://arxiv.org/abs/1909.08053, 2019.

[2] Nvidia collective communication library (nccl). https://developer.nvidia.com/nccl, 2023.

[3] Nvml api reference guide. https://docs.nvidia.com/deploy/nvml-api/index.html, 2023.

[4] ConnectX-6. https://www.nvidia.com/en-sg/networking/ethernet/connectx-6/, 2024.

[5] Cupti. https://developer.nvidia.com/cupti, 2024.

[6] Dynolog. https://github.com/facebookincubator/dynolog, 2024.

[7] Intel® performance counter monitor (intel® pcm). https://github.com/intel/pcm, 2024.

[8] Kineto. https://github.com/pytorch/kineto, 2024.

[9] Kubernetes. https://kubernetes.io/, 2024.

[10] Matrix Multiplication Background User's Guide. https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html, 2024.

[11] Median absolute deviation. https://en.wikipedia.org/wiki/Median_absolute_deviation, 2024.

[12] Mellanox/mstflint. https://github.com/Mellanox/mstflint, 2024.

[13] NVIDIA CONNECTX-7 400G ETHERNET SMART ACCELERATION FOR CLOUD, DATA-CENTER AND EDGE. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf, 2024.

[14] Nvidia dcgm documentation. https://docs.nvidia.com/datacenter/dcgm/latest/index.html, 2024.

[15] NVIDIA H100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/h100/, 2024.

[16] NVIDIA H800 PCIe 80 GB. https://www.techpowerup.com/gpu-specs/h800-pcie-80-gb.c4181, 2024.

[17] Nvidia nemo framework user guide. https://docs.nvidia.com/nemo-framework/user-guide/latest/overview.html, 2024.

[18] Nvidia nsight compute. https://developer.nvidia.com/nsight-compute, 2024.

[19] Nvidia nsight systems user guide. https://docs.nvidia.com/nsight-systems/UserGuide/index.html, 2024.

[20] NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/, 2024.

[21] Pytorch profiler. https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html, 2024.

[22] Slurm Workload Manager - Documentation. https://slurm.schedmd.com/documentation.html, 2024.

[23] bpftrace. https://bpftrace.org/, 2025.

[24] Dynamically program the kernel for efficient networking, observability, tracing, and security. https://ebpf.io/, 2025.

[25] Nccl profiler plugin documentation. https://github.com/NVIDIA/nccl/tree/master/ext-profiler, 2025.

[26] Pytorch/gloo. https://github.com/pytorch/gloo, 2025.

[27] Cursor: The best way to code with ai. https://cursor.com, 2026.

[28] Jax: High performance array computing — jax documentation. https://docs.jax.dev/en/latest/, 2026.

[29] Bishop, C. M., and Nasrabadi, N. M. *Pattern recognition and machine learning*, vol. 4. Springer, 2006.

[30] Comaniciu, D., and Meer, P. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on pattern analysis and machine intelligence 24*, 5 (2002), 603–619.

[31] Deng, Y., Zhang, L., Wang, Q., Zhi, X., Zhang, X., Jiang, Z., Xu, H., Wang, L., Song, Z., Liu, G., Bai, Y., Wang, S., Xiao, W., Ye, J., Yu, M., and Xu, H. Mycroft: Tracing dependencies in collective communication towards reliable llm training, 2025.

[32] Dong, J., Qian, K., Zhang, P., Zheng, Z., Chen, L., Feng, F., Xu, Y., Zhu, Y., Lu, G., Li, X., et al. Evolution of aegis: Fault diagnosis for {AI} model training service in production. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)* (2025).

[33] ESTER, M., KRIEGEL, H.-P., SANDER, J., XU, X., ET AL. A density-based algorithm for discovering clusters in large spatial databases with noise. In *kdd* (1996), vol. 96, pp. 226–231.

[34] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. Vl2: A scalable and flexible data center network. In *SIGCOMM* (2009).

[35] JIANG, Z., LIN, H., ZHONG, Y., HUANG, Q., CHEN, Y., ZHANG, Z., PENG, Y., LI, X., XIE, C., NONG, S., ET AL. Megascale: Scaling large language model training to more than 10,000 gpus. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (2024), pp. 745–760.

[36] LIU, K., JIANG, Z., ZHANG, J., GUO, S., ZHANG, X., BAI, Y., DONG, Y., LUO, F., ZHANG, Z., WANG, L., SHI, X., XU, H., BAI, Y., SONG, D., WEI, H., LI, B., PAN, Y., PAN, T., AND HUANG, T. R-pingmesh: A service-aware roce network monitoring and diagnostic system. In *Proceedings of the ACM SIGCOMM 2024 Conference* (New York, NY, USA, 2024), ACM SIGCOMM '24, Association for Computing Machinery, p. 554–567.

[37] MCINNES, L., HEALY, J., ASTELS, S., ET AL. hdbscan: Hierarchical density based clustering. *J. Open Source Softw. 2*, 11 (2017), 205.

[38] MILLS, D. L. Internet time synchronization: the network time protocol. *IEEE Transactions on communications 39*, 10 (1991), 1482–1493.

[39] QIAN, K., XI, Y., CAO, J., GAO, J., XU, Y., GUAN, Y., FU, B., SHI, X., ZHU, F., MIAO, R., WANG, C., WANG, P., ZHANG, P., ZENG, X., RUAN, E., YAO, Z., ZHAI, E., AND CAI, D. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference* (New York, NY, USA, 2024), ACM SIGCOMM '24, Association for Computing Machinery, p. 691–706.

[40] XIONG, Y., JIANG, Y., YANG, Z., QU, L., ZHAO, G., LIU, S., ZHONG, D., PINZUR, B., ZHANG, J., WANG, Y., JOSE, J., POURREZA, H., BAXTER, J., DATTA, K., RAM, P., MELTON, L., CHAU, J., CHENG, P., XIONG, Y., AND ZHOU, L. SuperBench: Improving cloud AI infrastructure reliability with proactive validation. In *2024 USENIX Annual Technical Conference (USENIX ATC'24)* (July 2024).

APPENDIX

A Case Study 4: Hardware Issues

A text-to-picture LMT job of 2,560 Nvidia H800 GPUs is expected to take 5 seconds per iteration, but is training at 9 seconds per iteration ("expected" and "original" in Fig. 18).

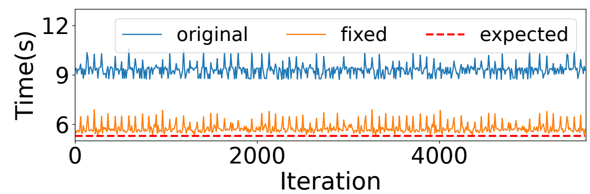
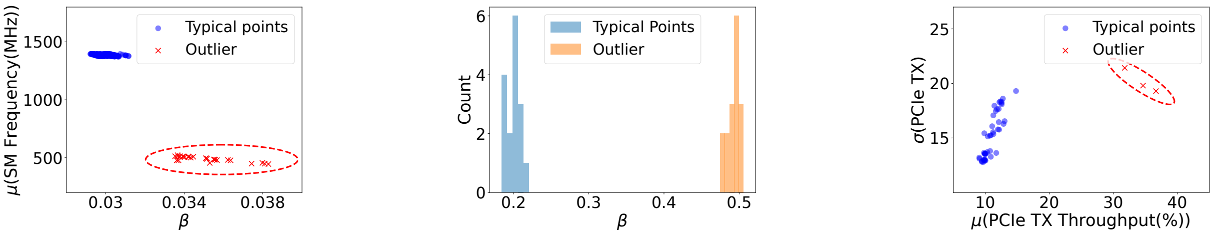


Figure 18: Iteration time of the LMT in Case Study 4.

Problem 1: GPU throttling. EROICA outputs 300+ workers (located in 40+ hosts) with abnormal execution behaviors on multiple GPU computation kernel functions. Figure 19a shows the example of `GEMM` (one of the GPU computation kernel functions). GPU computation kernels on these workers have much larger β and smaller μ than most workers, so they have high values of $\Delta_{f,w}$. This indicates that the GPUs of these workers are much slower, so they need a longer time for computation. Interestingly, EROICA conducted multiple profiles of this LMT, and in each profile, the slow GPU workers were not consistent, but they are intensively distributed among workers in certain racks, rather than evenly distributed among all workers. This indicates that GPUs from specific deployment batches have intermittent performance issues. Hardware checks revealed recent `GPU_NVSMI_HW_THROTTLE` alerts on some of these machines, confirming the risk of intermittent GPU throttling. We have contacted the hardware vendor for repairs on these machines.

Problem 2: NVLink down. EROICA also outputs 3 workers with abnormal execution behaviors on multiple collective communication functions. Taking `AllGather_RING` (one of the collective communication functions) as an example, 48 workers (belonging to three data-parallel groups, where each group has 16 workers) have significantly larger values of β than the other 2,512 (Figure 19b). Among these 48 workers, three of them have significantly larger μ than the other 45 (Figure 19c), indicating a higher bandwidth usage of PCIe. Hardware checks revealed NVLink Not Supported (NS) error on these three workers, which means all data flow sent to/from these three workers has to go through PCIe rather than NVLink. Since PCIe is much slower than NVLink, it leads to communication inefficiency.

Limitations of hardware monitors. Section 2.1 discusses limitations of hardware monitoring. In this case, for Problem 1, the affected GPU throttling events occur at low frequency with short durations (100us–10ms), failing to trigger Nvidia's built-in alerts. The monitoring system, operates at

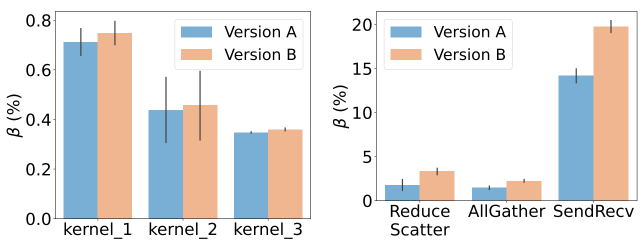


(a) The value of β and μ (average GPU SM frequency) of `GEMM` on 2,560 workers.

(b) β values of `AllGather_RING`. There are 2,512 typical points and 48 outliers. We sampled 16 for both of them.

(c) The value of μ and σ (PCIe TX throughput from GPU to NIC) of `AllGather_RING` on the 48 workers with high β value.

Figure 19: Function execution behaviors of `GEMM` and `AllGather_RING` in Case Study 3. Each point represents the runtime behavior pattern of one worker. Outliers are regarded as abnormal function executions with high $\Delta_{f,w}$ (§4.3) and are output by EROICA.



(a) β value comparison of three representative GPU computation kernel functions between Version A and B.

(b) β value comparison of representative collective communication functions between Version A and B.

Figure 20: β values of GPU computation and communication functions in Case Study 4 (we omit kernel names of GPU computation kernel functions since they are too long).

minute-level granularity (with second-level precision in short intervals) to be light-weighted, also failed to capture such fine-grained performance fluctuations. For Problem 2, the affected machines were newly added to the cluster, and the monitoring agent had not been updated, thus failing to detect the problem.

Fixes. We replaced the problematic hosts with standby hosts and restarted the LMT. Figure 18 shows the iteration time after the replacement (the "fixed" line).

B Case Study 5: Code Issue (EROICA Failed to diagnose)

A reinforcement learning LMT job of 8 Nvidia H800 GPUs (located at one host) is expected to take ~ 22 seconds per training iteration (about hundreds of commits before the current version, called Version A), but is training at ~ 26 seconds per training iteration (called version B).

EROICA's Diagnosis result. EROICA diagnosed both Version A and B, revealing that most GPU computation kernels and collective communications in Version B exhibited slightly higher β values than Version A (Fig. 20a and 20b). Since β represents the time proportion within a single iteration and Version B had longer iteration durations, this indicates that most computation and communication functions consumed more time in Version B. In addition, EROICA observed no significant difference in μ values between the two versions, which confirmed no hardware issues, indicating that Version B executed more workload on GPUs and networks during identical functions.

However, our customer confirmed that the workload of Version A and B were exactly the same.

The root cause. This LMT job utilizes reinforcement learning techniques to train the data. It has both training processes and inference processes. Inference processes should execute remotely but also persist in an idle state on the host due to developer oversight, running concurrently with training processes while repeatedly invoking `allgather` for synchronization. Inference processes utilize gloo [26] to execute `allgather` by default, which is based on TCP, so they do not influence the training performance. However, in one commit (among the hundreds of commits between Version A and B), it was changed to NCCL (based on CUDA) to pursue better performance. The side-effect is that it contended for GPU resources with training processes (NCCL requires GPU SMs). This led to lower performance of both GPU computation and communication of training processes.

Why EROICA failed and how the root cause is identified. As discussed in §7, for most performance issues, when the problematic function and its behavior are identified, it is very close to the root cause, or we can even directly derive the root cause. In this case, however, there were too many "problematic" functions (including most GPU computation and communication functions). Moreover, no one was aware that training and inference are executed in the same host (in separate processes) in accident. Since only training processes have performance issues, we overlooked employing EROICA to diagnose other processes (*e.g.*, the idle inference process) on the same host.

The root cause is finally identified by human power. A team of 20 engineers performed a binary search among hundreds of commits between Version A and B, identifying a performance difference between two adjacent commits. This localization process took one month.

Lessons and rethinking: Potential opportunities to auto-

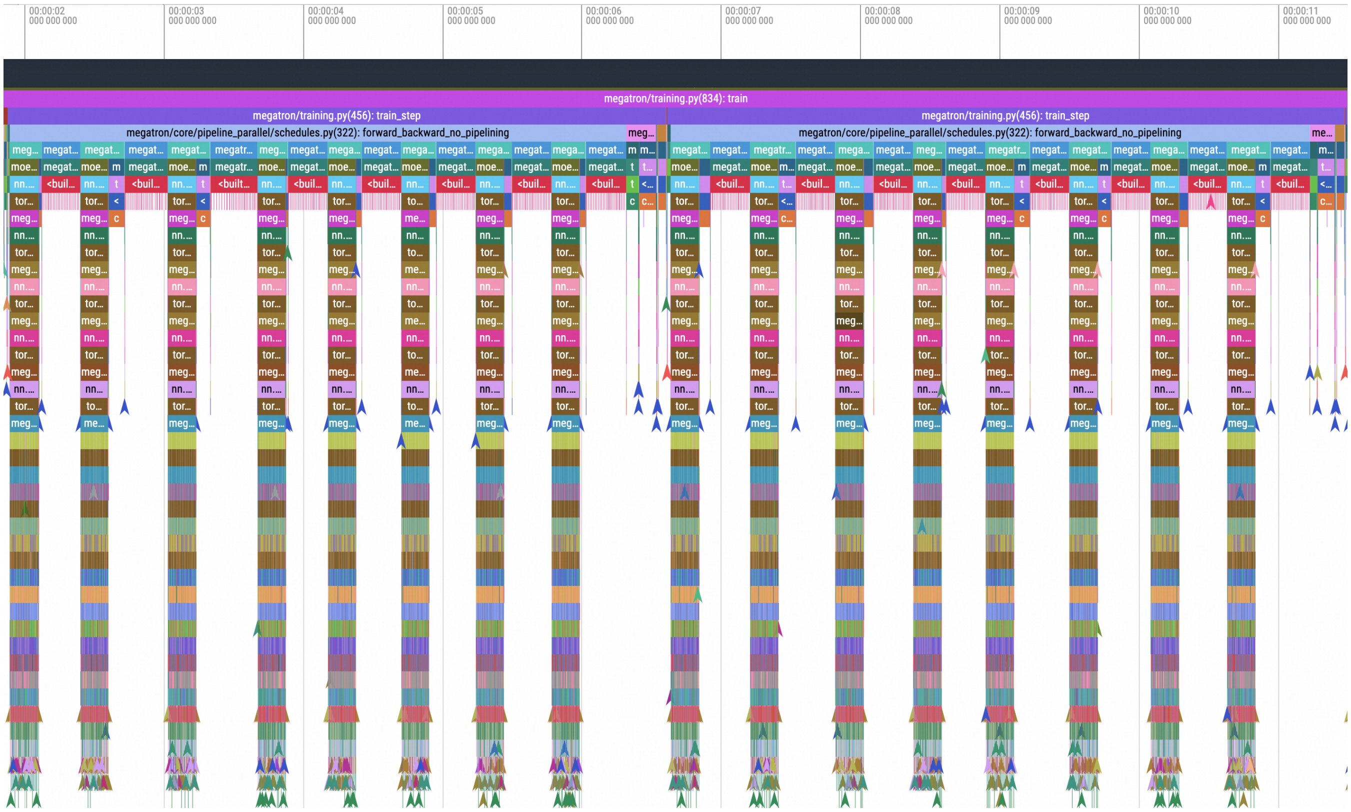


Figure 21: The runtime behavior of an MoE training.

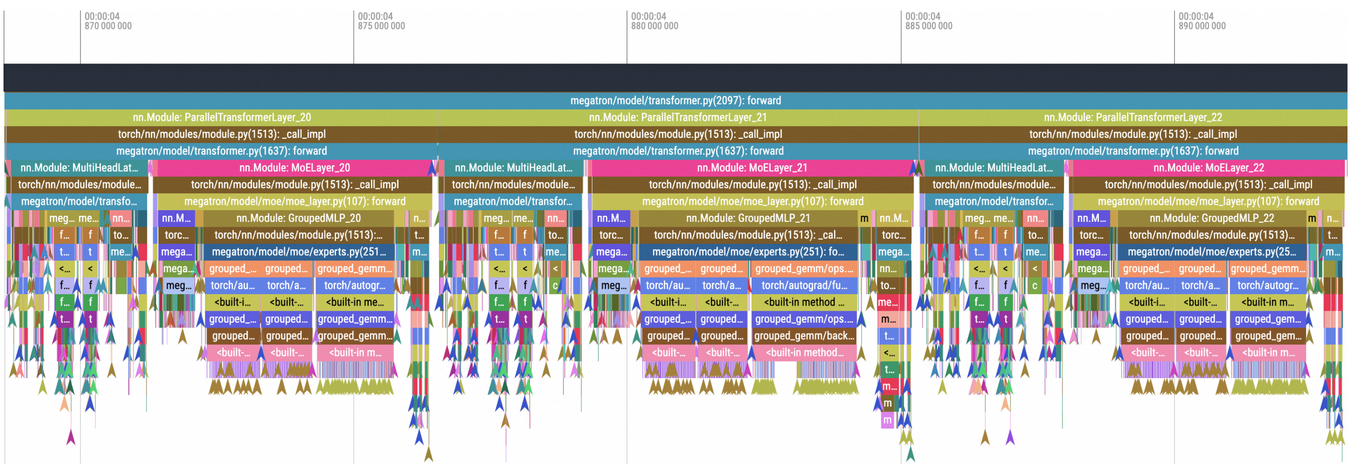


Figure 22: The python function behaviors of a forward step in the MoE training.

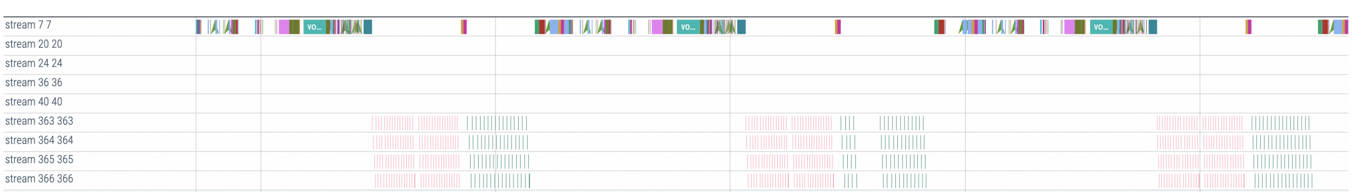


Figure 23: The kernel function (includes both computation and communication) behaviors of a forward step in the MoE training.

matically diagnose this case. As reinforcement learning becomes a promising technique of LMT, some LMT jobs execute processes of multiple actors in one host, like training and inference.

Based on the identified root cause, we retrospectively analyzed the diagnostic process. EROICA had at least two opportunities to successfully locate this issue: (1) We should first make an overview at all processes running on the host. Upon recognizing the potential concurrency between the (idle) inference and training processes in this LMT, EROICA should have been deployed to diagnose the idle inference process also, which would have directly revealed the NCCL function calls in the inference process. (2) Even based solely on training process diagnostics, the root cause could have been indirectly deduced. EROICA's detection of heavier workload execution in Version B versus Version A functions indicated resource contention.

The failure in problem localization primarily originated from our lack of an overview of the LMT, causing us to overlook the impact of other processes (even existing accidentally) on the training procedure. For future performance diagnostics, especially for the LMT with complex inter-process relationships, EROICA will implement usability optimizations, such as automatically expanding the diagnosis scope to all LMT-related processes within the host.

C Comparisons with State of the Arts

We compare EROICA with recent monitoring techniques [23, 25, 35] and profilers; Table 3 summarizes the results.

For online monitors, MegaScale [35] deploys techniques to record CUDA-event timelines (to expose slow GPU kernels) and perform ms–s RDMA monitoring, but mainly report alerts/statistics; root-cause localization (especially for network issues) is typically manual. It also lacks Python function events, and thus cannot diagnose code-level issues. NCCL Profiler targets only communication by instrumenting the communication library. Bpftrace [23] selectively instruments key Python functions/kernels (*e.g.*, via .so replacement) to detect slowdowns, but root-cause analysis still often requires manual effort. We also compare against directly using Nsight Systems [19] and Torch Profiler [21]. Although they support offline/custom analysis (Nsight's Post-Collection Analysis API and Torch's JSON dumps), running them online incurs prohibitive overhead in production LMT.

Table 3 further compares troubleshooting ability and diagnosis time on the problems in §6.1 and §6.2 (*e.g.*, Case1-P1 represents Problem 1 of Case Study 1). Existing online monitors miss many issues due to incomplete data sources. Profiler traces cover more, but loading data from all workers can take days; since they do not provide built-in troubleshooting logic, we report only the data-loading time (end-to-end diagnosis would take longer).

Table 3: Comparisons with state-of-the-art approaches

Technique	Case 1			Case 2				Diagnostic time for 10,000-GPU LMT
	P1	P2	P3	P1	P2	P3	P4	
MegaScale [35]	✗	✗	✗	✗	✓	✗	✗	online
NCCL Profiler [25]	✗	✗	✗	✗	✓	✗	✗	online
bpftrace [23]	✓	✗	✓	✗	✗	✗	✗	online
Nsight System	✗	✗	✗	✓	✓	✗	✓	>1.5 days (offline)
Torch Profiler	✓	✓	✓	✗	✗	✓	✓	>3.5 days (offline)
EROICA	✓	✓	✓	✓	✓	✓	✓	3 min (online)

Table 4: Overhead in different model configurations

Configurations			Overhead		
model size	tp	pp	training (s/iter)	profiling (s/iter)	generate data (s)
gpt3-7b	1	1	1.371	1.389	13
	2	1	1.777	2.002(+12%)	19
gpt3-13b	2	1	2.489	2.485	17
	4	1	1.228	1.425(+16%)	26
	8	1	1.528	1.707(+11%)	28
gpt3-65b	8	4	1.191	1.202	15
	8	8	1.281	1.288	10

D Profiling Overhead in Different Model Configurations

As explained in §6.4, with EROICA, profiling and data generation are the two components executed in the main process of LMT (Figure 16), Table 4 evaluates them systematically in different LMT configurations. In general, the time spent on data generation correlates positively with the number of function executions on a single worker within the profiling window. Increasing the degree of parallelism (*e.g.*, the tensor parallelism, TP) results in more fragmented model training. This fragmentation leads to more function execution events, increasing the data generation time.

In most configurations, profiling does not introduce overhead to training. However, when the model is small and the parallelism parameters are large (*e.g.*, GPT-3 7B with TP=2; GPT-3 13B with TP=4 or 8), there is a major increase in training time. This is because such configurations impose a higher load on the CPU; as profiling requires CPU computation, it causes contention. Fortunately, these impractical configurations are never used in production. Note: EROICA is still effective in these configurations and the overhead only happens during the profiling window.

E Examples of LMT executions

Figure 21 shows the timeline (using https://ui.perfetto.dev/) of high-level Python function call stacks of two adjacent training iterations of one of our production MoE models. Each iteration includes multiple forward and backward phases, and their function runtime behaviors are identical. All workers are executing the same functions with the same behaviors. Due to space limitations, we only present one of them.

Figure 22 and 23 zoom into one of the forward phase in Figure 21. It shows that in the forward phase, low-level functions are also executed repeatedly, and their behaviors (*e.g.*, duration of execution) are highly similar across multiple executions.

In large model training, especially in heterogeneous computation, workers may not exhibit identical execution behavior due to differences in parallelization roles (*e.g.*, pipeline/expert parallelism), input variability (*e.g.*, variable-length video samples), and system effects (*e.g.*, scheduling and contention). Nevertheless, worker-side execution is usually structured as repeated iterations that invoke a largely stable set of training-step functions and GPU kernels. Moreover, tensor shapes are often fixed by model configuration and batching policy, or are restricted to a small set of discrete shapes through padding, bucketing, truncation, or shape-constrained micro-batching. As a result, per-function runtime and resource-usage metrics across workers are expected to be either broadly consistent or, when not identical, to follow a relatively stable distribution reflecting systematic role differences and stochastic runtime noise. Under this assumption, function instances that fall outside the expected cross-worker distribution—*i.e.*, statistically significant outliers in runtime or resource consumption—can be treated as anomalous behavior.