THORN-ML: Transparent Hardware Offloaded Resilient Networks for RDMA based Distributed ML Workloads

Maziyar Nazari, Daniel Noland†, Giulio Sidoretti, Erika Hunhoff,
Tamara Silbergleit Lehman, Eric Keller
{first.last}@colorado.edu
University of Colorado Boulder, † Unaffiliated

Abstract

Distributed deep learning (DDL) requires a great investment in cloud infrastructure, including accelerated compute nodes and networking hardware capable of supporting high-performance networking, e.g., Remote Direct Memory Access (RDMA). When a host running a DDL application becomes unreachable, the cost can be high as application-level failure recovery is slow and disruptive. When the host is unreachable due to host failure, this is unavoidable; however, when the network components involved in attaching the host to the core data center network fail, we argue that this cost is avoidable. This paper introduces THORN-ML, a hardware-offloaded resilient network architecture that is completely transparent to DDL applications and works with commodity hardware. We evaluate THORN-ML on a cluster of 5 nodes with Nvidia A100 GPUs and Mellanox ConnectX-5 NICs, with several applications leveraging model parallelism and/or data parallelism, and find that THORN-ML reduces disruption from minutes (impacting the whole cluster) to milliseconds (impacting packets that can be re-transmitted).

CCS Concepts

• **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Networks** → **Data center networks**; • **Hardware** → *Networking hardware*; • **Computing methodologies** → **Machine learning**; **Artificial intelligence**.

Keywords

RDMA, Distributed Deep Learning, SmartNIC, Cloud Computing, Resilient, Checkpointing, ML, AI

ACM Reference Format:
Maziyar Nazari, Daniel Noland†, Giulio Sidoretti, Erika Hunhoff,, Tamara Silbergleit Lehman, Eric Keller. 2025. THORN-ML: Transparent Hardware Offloaded Resilient Networks for RDMA based Distributed ML Workloads. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3772052.3772225

1 Introduction

Deep learning has seen tremendous growth, stemming from both advances in algorithms and advances in infrastructure—both of which support ever-increasing sizes of models and training data. This application is the current major driver of growth in cloud computing—whether traditional cloud providers (e.g., AWS, Azure, GCP), private clouds, or a new class of hyperscaler [7].

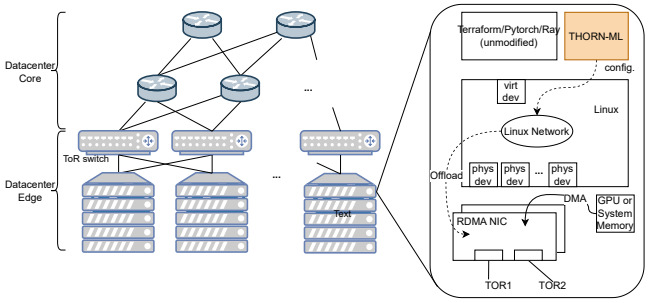

Figure 1: The left is an example data center architecture with added network redundancy at the datacenter edge. On the right is a zoomed in view of one representative server in the data center. The server view illustrates a DDL application with THORN-ML running in user space. THORN-ML which sets up a virtual device for the application, performs routing between the virtual device and physical devices, performs fault detection, and supports the offload of the Linux configuration to RDMA NIC hardware.

The clusters which run distributed deep learning (DDL) applications consist of racks of servers, each with network cards (NICs) connected to a top-of-rack (ToR) switch, which are then interconnected through a topology of core network switches (Figure 1). As a snapshot of the scale of this infrastructure, Meta invested in a cluster with 24,000 GPUs, with the goal (at the time) of managing 350,000 total GPUs by the end of 2024 [35], updated only a few months later to 600,000 [49], and presumably even higher today.

Unfortunately, there is a significant reliability problem that exists, that is only going to grow more problematic in the future as the clusters used for DDL training increase in scale. DDL frameworks, such as TensorFlow [47], Ray [19], and PyTorch [41], handle host-level failures by using distributed checkpoints. Training typically involves several epochs of processing where forward and backward propagation occur in neural networks for a batch of data. A checkpoint of the state of the application (which is distributed across the cluster) is then recorded at specified intervals.

In the event of a failure, the DDL training job can restart from the last checkpoint, avoiding the need to start from scratch. This mechanism, however, causes a significant disruption, as all workers must revert to the checkpoint. In practice, checkpoint frequencies of 20 to 30 minutes are common. That means that in the case of a failure, there will be tens of minutes in which the cluster did not do useful work. This is quite costly, in terms of power consumption to run a large scale cluster, opportunity cost for not being able to use

SoCC '25, Online, USA

ACM ISBN 979-8-4007-2276-9/25/11
https://doi.org/10.1145/3772052.3772225

the cluster for another job, and developer time for those who are waiting on the job's completion. Even more, there are some cases where the training job is unable to make any progress if the time between any failures is less than the checkpointing period (e.g., consider a link flapping up and down).

Currently, it is deemed impractical to checkpoint more frequently, but, to get a sense of the most optimistic possibility, we performed a measurement of the absolute minimum downtime. In our measurements on a 5 node GPU cluster, training the GPT-2 model and ResNet101, a single epoch takes anywhere from 4 to 80 seconds, and the time to initialize all nodes running workers takes around 21 seconds (details in Section 2). Combined, this is the time for which the resources reserved for this job cannot perform any useful computation. Our measurements are likely greatly underestimating the overheads in practice as they were gathered on a small cluster, but likely to be notably higher on bigger clusters. Further, these time estimates do not include detection time, the time to transfer from a storage server, or time to initialize a new worker node if the network failure is not transient.

Core data center networks are designed with redundancy and routing protocols to detect and route traffic around failures with little disruption [30, 33, 40]. However, this redundancy does not extend to the edge of the datacenter network, leaving the connectivity between the host and the Top-of-Rack (ToR) switch vulnerable to failures (accounting for nearly 10% of overall disruptions, based on measurements from Meta [29].). The components at this connection point include the NIC on each host, the ToR switches themselves, and the cables that connect the NICs to the ToR switches. If any of these components fail, current practice detects the hosts as unreachable and treats the failure the same as a whole server failure (triggering the rollback to a checkpoint).

The increasing scale of infrastructure used for DDL training has an unfortunate double impact affecting reliability. The large (and increasing) number of components (servers, switches, etc.) involved in a single DDL training job, *increases the probability of failure* of the cluster experiencing a failure of a component. At the same time, the increasing size of a cluster *increases the impact of a failure*, where all servers need to be reverted to a checkpoint, wasting the expensive GPU cycles and power consumption of the entire cluster.

Given the associated impact of failure, one might ask why the redundancy found in the rest of the datacenter network is not extended into the datacenter network edge. The reason is not necessarily cost, but the fact that distributed deep learning applications would not even be able to take advantage of it without significant work. The problem is due to the widespread use of Remote Direct Memory Access (RDMA) to transfer data in distributed deep learning infrastructures, which enables low-latency, high-throughput transfers between servers without CPU involvement. With RDMA, the programming abstraction inherently assumes that an application binds to the device it is going to use to make transfers. Further, there is no mechanism to detect if a component at the datacenter network edge has failed to work around it.

In this paper, we introduce THORN-ML, an architecture for resilience at the datacenter network edge that requires no modifications to applications (or libraries) and works with commodity hardware and standard protocols. THORN-ML's role in the datacenter is depicted in Figure 1, The datacenter network is modified to add an additional (redundant) connection from each host to a different ToR switch. THORN-ML then runs alongside the application on each host. This approach enables applications to continue making forward progress without having to restart from a checkpoint when failures are encountered in components at the edge of the datacenter network.

The key in THORN-ML to supporting unmodified applications is the introduction of a single virtual network device[1] that the application binds to. We then extend the network routing layer in the host, using routing software, to provide failure detection and recovery that is aware of the virtualization and available redundancy (the added connection to a different ToR switch). Running alongside the application makes this more deployable and principled in its design, requiring no modifications to the application or one of the many possible libraries it uses [11, 13, 20, 39, 52].

We take advantage of the programmability of commodity NICs to make THORN-ML more deployable. We leverage the Linux switchdev driver by forming Linux traffic control (*tc*) rules that are offloaded to hardware. These rules require a translation layer between the state produced by the routing software (FRR [9], Bird [27], GoBGP [12], etc.) and what the switchdev driver can offload. Given that Netlink, the protocol to monitor the Linux kernel, does not guarantee delivery of updates, we introduce a reconciliation loop to ensure correctness.

We evaluate a prototype of THORN-ML on a cluster of five servers, each with an Nvidia A100 GPU (supporting GPU Direct [18]) and a Mellanox ConnectX-5 NIC (dual 100Gbps port), connected through two ToR switches. We show: (1) that THORN-ML supports unmodified applications, tested with GPT-2, ResNet101, and the Nvidia Collective Communication Library (NCCL), (2) that there is no noticeable impact in the time to accuracy for applications even with an aggressive failure model (3) and that RDMA traffic performance tests incur a nominal decrease in bandwidth and increase in latency. Together, the minimal overhead and the ability to eliminate the unnecessary cost of recomputation in the case of edge network failures demonstrate the use and practicality of THORN-ML.

In summary, we make the following contributions:

- Measurements to quantify the minimum impact of a failure in one component, and background that overviews why this is a challenge to solve with RDMA (Section 2).
- Introduction of user space software that can be run along side an unmodified application, that provides all of the needed functionality to capitalize on added redundancy at the network edge (Section 4)
- An approach for interfacing with Linux networking to take advantage of hardware offload capabilities of commodity hardware, such that there is no overhead in RDMA data transfers (Section 5).
- An evaluation on a real cluster with GPUs, with a variety of real unmodified DDL applications, demonstrating the ability to handle even an aggressive failure model without disruption (Section 6).
- Code is available at https://github.com/cu-ml-net

[1] A network device in Linux is used to represent, among other things, each individual network port in a NIC—e.g., eth0, eth1.

2 Motivation

In current architectures, DDL frameworks cannot distinguish between a failure of the host itself or components that connect the host to the core datacenter network. For example, if a cable between the host and the ToR switch is cut, that host will be detected as unreachable, but the detection mechanism does not know if it was a faulty cable or a downed host. As such, it is natural for these frameworks to handle failure similarly.

In this paper, we advocate for redundancy at the network edge, such as a NIC with two ports each connected to a different ToR switch (as illustrated in Figure 1). We argue that this approach is better than application-level fault handling mechanisms (Section 2.1), but hard to support transparently (Section 2.2).

2.1 Need for Network Edge Resilience

To understand the degree to which fault handling can be improved by resilience at the network edge (as provided with THORN-ML), we explore both the overhead of current fault handling techniques and posit why these overheads are expected to grow.

2.1.1 How Host Failure is Handled. Due to the increasing volume of data and the expanding size of deep learning models, DDL has become essential in the current era of machine learning. DDL training is a parallel computing application composed of a number of workers, deployed on a set of nodes, that perform computational tasks and periodically communicate with each other (commonly through collective operations). Modern frameworks employ various forms of parallelism, such as data and model parallelism, to distribute the training workload across clusters of machines.

Training is a single application working towards a single output, i.e., a trained model. With this approach, the 'global state' of an application is distributed and there is inherent dependence between nodes through the communication that occurs. Failure of one node loses part of that global state. Due to the dependencies between nodes, we cannot just arbitrarily restart a single node; this limitation is a well established problem [34].

Popular DDL frameworks (Pytorch, TensorFlow, Ray) handle node failures through distributed checkpoints [22, 23, 28]. Developers can configure the frequency of when a checkpoint is taken, typically defined in terms of number of epochs (i.e., iterations where forward and backward propagation occur on a batch of data in neural networks).

When failure is detected, the application must be restarted from the last checkpoint. This applies to all workers, not just the node that failed, because a checkpoint represents the global state for the distributed application.

2.1.2 Impact of Failure. The impact of a failure is largely determined by the checkpointing interval. A longer interval will reduce the overheads of taking the checkpoints, but increase the amount of wasted compute time when failure occurs. As a starting point, the default interval time for performing checkpoints in Tensorflow is 10 minutes [25]. In our conversations with industry experts, 20-30 minute intervals are commonly used. For large training jobs / infrastructures, it is not practical to checkpoint more frequently. When taking a checkpoint, each node needs to transmit its state across a network to a set of storage servers. This leads to significant overheads of extra compute, network bandwidth, and storage server I/O.

Table 1: Checkpointing and initialization latency/overhead in GPT-2 training on CodeParrot dataset using Megatron-LM and ResNet101 training on CIFAR-10 dataset using Pytorch DDP, along with epoch time for each. The model architecture is annotated as Pipeline Parallel (PP), Data Parallel (DP), and/or Tensor Parallel (TP). Measurements are in seconds.

Model Arch.	Avg Checkpoint Latency	Model Init. Latency	Avg Epoch Time
GPT-2 (DP)	0.771	22.611	4.4
GPT-2 (PP)	0.534	21.283	8.1
GPT-2 (PP+TP)	0.342	20.912	9.9
ResNet101 (DP)	0.240	5.693	80.0

To understand the impact of checkpointing frequency we perform a study in our in-house small cluster. We take measurements on a cluster (fully described in Section 6) with 5 hosts, each with a GPU and dual port 100Gbps NIC, connected through two switches, along with a high performance storage server also connected to the switches. There are three components to fault handling: (1) the time to perform a checkpoint – which adds time to each training iteration, (2) the model initialization time – which is needed on each node when a re-start is required after a failure, and (3) epoch time – which determines the minimum period for taking checkpoints. We measure the effects of failure for two training tasks: GPT-2 [5] (with three different architectures) and ResNet101 [26].

As shown in Table 1, each epoch in GPT-2 training using Megatron-LM [45] (framework for training large-scale language models) takes 4.4, 8.1 and 9.9 seconds on average for data parallel (DP), pipeline parallel (PP) and tensor-pipeline parallel (PP+TP) configurations respectively. Initialization time after a failure introduces an overhead of 20.9-22.6 seconds for those same three configurations. This results in 27.0-30.8 seconds (adding up the initialization latency and the average epoch time) of unnecessary delay per failure, when checkpointing after every epoch. For ResNet101 training on the CIFAR-10 [4] dataset using PyTorch [41] Distributed Data Parallel (DP), this overhead can reach upwards of 85 seconds.

These numbers assume checkpointing at every epoch, which is not common in real-world scenarios. Additionally, these figures represent the overheads for a 5 node cluster, whereas a larger cluster likely exhibits notably higher numbers due to increased I/O.

2.1.3 Frequency of Failure (is Growing). It is useful to understand the extent of the costs of edge failures that THORN-ML seeks to address. In a recent paper discussing Llama 3 models, Meta disclosed some operational data on their 16,000 GPU cluster over a 54-day snapshot [29]. Of the 466 job interruptions recorded, 8.4% were due to a network switch or cable failure, and an additional 1.7% were due to NIC failure. While not all interruptions can be attributed to the network edge, we argue that this is still a substantial disruption – especially given the impact (and cost) a failure can have on training.

Increasing model sizes (approaching 10^{26} today, growing at 4.1x per year [42]) and training dataset sizes (at 10^{12} samples, growing at 0.11-0.23 orders of magnitude per year [50]), require larger and larger networks. The large number of components in the system increases the likelihood of failure of individual components while simultaneously increasing the cost of a failure for any idle cycles due to failure.

2.1.4 Maintenance is Planned Failure. The above refers to network component failure, but maintenance, such as replacing an optic cable, is another form of failure that is quite impactful. The impact to training jobs can many times be avoided, with careful planning. But, this planning increases the complexity of network operations – which commonly encompasses a large fraction of network costs. With an architecture that can seamlessly handle failures, network management becomes greatly simplified since operators can perform the maintenance needed without worrying about the impact to the workloads.

2.2 Challenge with Redundancy

We posit that handling network faults should be transparent to the application and should not require the development of custom hardware or network protocols. This subsection discusses why the use of RDMA in large-scale DDL poses a challenge to application transparent failure handling.

2.2.1 Remote Direct Memory Access (RDMA) Background. RDMA enables the transfer of data between memory in two different servers without the involvement of the CPU in either server [43, 44]. The memory can be system memory or the GPU's memory (through GPU Direct [18]). This protocol facilitates efficient and effective communication, as CPU cycles are reserved for application processing and the data transfer is not bottlenecked by the CPU. RDMA was originally designed to work over Inifiniband, but is now widely supported over Ethernet fabrics through RDMA over Converged Ethernet (RoCE).

Consider Figure 2, which illustrates how RDMA transfers data between Node 1 and Node 2. The first thing to note is that each node has a send and receive queue (referred to as a queue pair). These queues are for applications to post work requests that detail information about the transfer (such as the memory buffers holding the data to be sent, or buffers for where to place received data). Step ①a and ①b illustrate this process, with the work requests stored as Work Queue Elements (WQE) in the respective queues. At this point the requests are cached in the NIC memory (steps ②a and ②b). With that information, Node 1's NIC (the sender) creates a direct memory access (DMA) transfer from the application's memory to the NIC (step ③), leveraging the Memory Translation Table (MTT) on the NIC. Node 1's NIC then forms this data into RDMA messages and transmits them (step ④). Node 2's NIC acknowledges the transfers (step ⑤) as part of the reliable connection. RDMA has two modes of transfers, unreliable and reliably connected. We assume the reliably connected mode, where hardware ensures data transfer is successful, since it is the more commonly used. The Queue Pair Context (QPC) on the NIC maintains the state of the transfer (DMA state, and connection state). Note, if messages need to be re-transmitted (not shown) it occurs here. At this point, Node

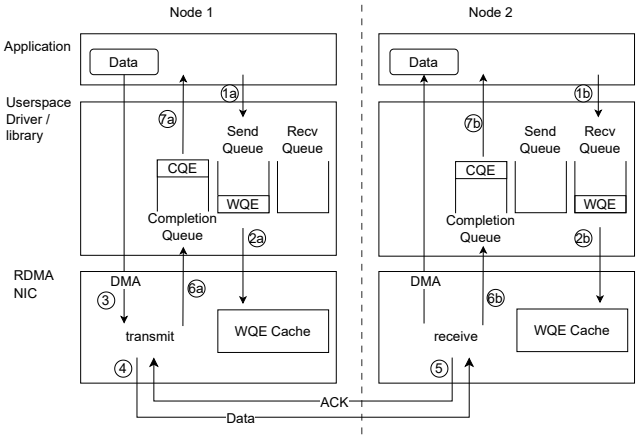

Figure 2: Overview of an RDMA transfer, where Node 1 is sending to Node 2.

1's NIC has sent the data and received an ACK, and Node 2's NIC has received the data. Each puts a Completion Queue Element (CQE) into the respective completion queues (steps ⑥a and ⑥b), which notifies the application (steps ⑦a and ⑦b) that the work has been completed.

2.2.2 Application Modifications. The programming model of RDMA is inherently tied to hardware as it involves setting up memory regions and context for the transfer (e.g., the destination host). The NIC creates RDMA control messages to establish the context, then the application can use the *libibverbs* API to set up queue pairs. The *libibverbs* API encapsulates the functionality needed to set up the hardware to initiate a transfer. The first call made by an application is to get a list of devices which support RDMA (`ibv_get_device_list()`). Then uses the corresponding verb to open one of those devices (`ibv_open_device()`). Each port of a NIC appears in Linux as a separate device. Applications can then set up a protection domain (`ibv_alloc_pd()`) and completion queue (`ibv_create_cq()`) for the specific device. Once the protection domain is setup, the application can register the memory region to allow the device to read/write data to this memory (`ibv_reg_mr()`) and setup its queue pairs (`ibv_create_qp()`). When setting up a queue pair, the application indicates the connection type, and in many cases, including DDL, this will be the 'reliable connection' to enable the hardware to track lost packets and perform re-transmission.

When an application uses RDMA, it is tightly coupled with the underlying device (i.e., a specific network port). If a failure occurs, the application (or integrated library) is required to detect network failures (network awareness), choose an available path (path discovery), and change the device used for the connection (re-initialize the queue pairs and communication channels).

2.2.3 Hardware and Protocol Modifications. Rather than modifying applications, there is a line of work that introduces new hardware and protocols that may support resilience at the network edge. These works are focused on addressing network congestion by proposing alternatives to equal cost multipath (ECMP) as they find that ECMP for ML workloads leads to congestion [32, 37, 46, 48].

Most of the proposed changes in prior work are to network switches, and do not address faults at the network edge [46, 48]. However, in Multi-Path RDMA (MP-RDMA), the authors propose a custom FPGA NIC to perform network load balancing [37]. The FPGA NIC performs congestion-aware packet distribution to multiple paths using one congestion window for all paths. A specialized path selection algorithm sends traffic on paths with similar delays to reduce out-of-order packets. MP-RDMA then sets the header based on desired path so that the ToR switch can forward along different paths. Given this characteristic, fault tolerance is considered between the ToR switch and the spine switch, but not at the datacenter network edge, the connection between the NIC and the ToR switch. MP-RDMA requires extending the RoCE protocol with new fields in the packet header to detect congestion, and requires all nodes to support the new protocol and hardware.

Given that this approach does not consider the network edge components, it is not suitable (without extension) to solve the need for resilience at the edge. Even if extended, it would face deployment challenges as it requires both custom hardware and a custom protocol.

3 Architecture Overview

Because there is not a readily available network-based solution to handle faults at the datacenter network edge today, application-level fault handling is used in distributed deep learning applications. Unfortunately, application-level fault handling is costly as it results in repeated work that wastes both expensive compute resources (namely GPUs) and power consumption. This fault handling also makes network management and operations more complex, as the impact of needing to perform maintenance is significant. Coordination is necessary to be able to take servers or network infrastructure offline between training jobs.

THORN-ML is a host-level network architecture that transparently increases resilience to failures at the network edge. In this paper we focus on the problem of supporting redundancy across two ports in a single NIC, and discuss supporting multiple NICs in Section 8. Figure 3, shows a detailed architectural overview to illustrate all of the pieces together. THORN-ML is composed of two main mechanisms to achieve failure resilience for unmodified applications, hardware, and protocols. We briefly overview these mechanisms in this section, and provide a detailed description of each in subsequent sections.

Unmodified application (Section 4): THORN-ML, enables redundancy without requiring that applications become aware of the underlying hardware to detect and handle failures. In RDMA-based networks, applications bind to a given network device and leverage a low-level protocol to set up communication channels with another host's network device. The key in THORN-ML to supporting transparency is that we introduce a single virtual device that the application will bind to. We couple the virtual device with an extension of the network routing layer to the host, with routing software and failure detection that is aware of the redundancy and sets routes in Linux. Most of these components leverage existing libraries and data structures inside the Linux Kernel, with some interfaces exposed to the user space (as illustrated in the red highlighted right side of Figure 3).

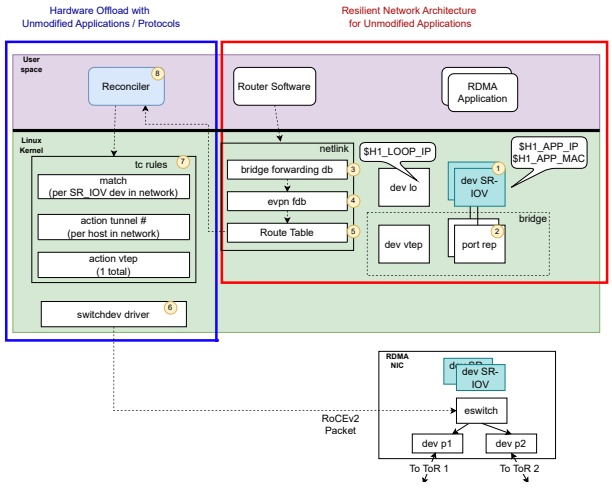

Figure 3: THORN-ML detailed architecture.

Unmodified hardware and protocols (Section 5): In RDMA-based networks, the NIC hardware plays a major role. To achieve data transfers between memory without CPU involvement, the NIC on each side of the transmission sets up a communication channel, and then the NIC is able to directly access a region of its own local system memory (or GPU memory in GPU Direct [18]) that it can form into packets and transmit (and re-transmit as needed). We solve the problem of resilience in THORN-ML by introducing redundancy using commodity hardware (without modifications). We take advantage of the increasing programmability of NICs, and through the support of the Linux switchdev driver where we can form Linux *tc* rules, which get offloaded to hardware. This requires a translation layer between the routing software and hardware offloadable rules, which we realize in user space and interface with the kernel through the Netlink protocol. To account for the fact that Netlink is a best effort protocol (does not guarantee delivery of updates), we introduce a reconciliation loop to ensure correctness. The reconciliation loop runs in user space and all other components are offloaded to the Smart NIC through the Linux Kernel existing interfaces (illustrated in the blue highlighted left side of Figure 3).

4 Resilient Network Architecture for Unmodified Applications

THORN-ML is a host-level network architecture that transparently increases resilience to failures at the datacenter network edge. Effectively, the goal is to provide applications with the abstraction of a Single Big Switch where each of the applications running on a host has a single network device that attaches to a switch. What is unique about THORN-ML is that the Single Big switch is highly available and works seamlessly with RDMA's unique properties.

This section describes three mechanisms THORN-ML incorporates to provide non-disruptive resilience for DDL applications using RDMA without requiring the application to be modified. To provide an intuitive overview, these three aspects can be illustrated in an abstracted view of the communication between two distributed instances of the application on different servers as

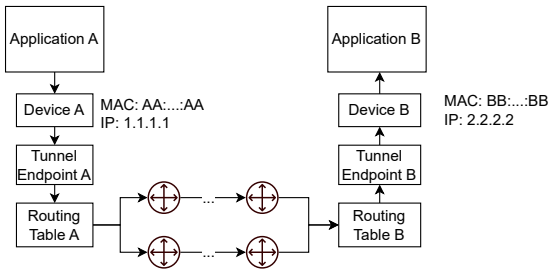

Figure 4: High-level packet walk in THORN-ML.

shown in Figure 4 (with the specifics captured in Figure 3). An application component (App A) needs to communicate with another component (App B) across the network. Its view is a simple connection to a single big switch that is highly available. Each application, App A and App B, has a network device (Device A and Device B respectively), with a MAC and IP address. This device is RDMA capable so that the application can bind to it. The applications use these devices, and their addressing mechanism, to communicate. Under the hood, and hidden from the application, the device has redundancy where traffic can take multiple paths.

To provide redundancy in the device, transparent to the application, routing software works in conjunction with both tunneling and routing tables to quickly detect and react to failure. Traffic that is sent through one of these devices is tunneled through the use of a Tunnel Endpoint (A and B in the figure). This tunnel encapsulates the traffic with server-level addressing to cross the network. Finally, the packet is subject to the routing table, which has each of the next hops from the redundant links as options in the table. This approach also allows THORN-ML to scale to arbitrarily large networks, as traffic is tunneled through this process with small overhead. All steps are designed to be offloaded to hardware to maintain the performance profile of RDMA (discussed further in Section 5).

4.1 Single Virtual Device

As covered in Section 2.2, applications are tightly coupled with the underlying device due to the programming model of RDMA. Rather than force applications to change how they interact with the RDMA hardware and become network aware, or change the underlying hardware and introduce extensions to RDMA, we introduce a novel approach which is transparent to applications and works with commodity hardware. THORN-ML creates a single (virtual) device that an application can bind to that can direct traffic to any underlying (physical) RDMA port, whether to different ports on the same NIC or to ports on different NICs.

To create this separation between the hardware and the software, we leverage SR-IOV, an I/O virtualization technology commonly supported in NICs [24]. Setting up an SR-IOV device results in two devices in Linux: the SR-IOV device, and a port representor (① and ② illustrated in Figure 3 respectively). The SR-IOV device is a physical device that implements the functions of the underlying NIC. The port representor represents a virtual port and can be interacted with using the standard Linux network stack [16].

The SR-IOV device is assigned both a MAC address and an IP address, which we call H1_APP_MAC and H1_APP_IP—H1 to indicate this address is for host 1, APP to indicate what that this is what the application sees, and IP/MAC to indicate the type of address. A DDL application can connect to the equivalent APP addresses for different workers.

Multiple SR-IOV devices may be created. One device can be created for each worker running on a host and others created for other application infrastructure, such as connections to storage or databases containing training data; THORN-ML can provide resilience for accessing those services as well.

4.2 Bridging to a VXLAN Tunnel

With applications now binding to the SR-IOV devices, THORN-ML must create a network tunnel between SR-IOV devices on different workers in order to hide the underlying network redundancy. We use VXLAN to create the network tunnels, which is commonly used in datacenters and multi-tenant cloud environments. VXLAN creates a single layer 2 network regardless of the underlying network protocols [38].

VXLAN allows us to create a single virtual tunnel end point (VTEP) device in Linux with a shared VXLAN network identifier (VNI) instead of creating individual tunnels between every pair of end points. In a multi-tenant or multi-application environment, different VNIs can be used to create multiple isolated virtual networks.

THORN-ML must make it appear as if all of the SR-IOV devices for all of the workers (or databases, etc.) are directly attached through a single big switch. To do this, we put the SR-IOV devices and the VTEP device into a single Linux bridge (composed of items ③, ④ and ⑤ shown in Figure 3). The VTEP is used for application-to-application addressing.

To also support host-to-host addressing, we assign an IP address to the loopback device (lo) on each host, which is a device that is always up and can be used to identify the host consistently across all network devices with a single IP address. Once the VTEP is created, we can configure Linux tables to encapsulate and direct traffic. Figure 5 shows an example of a complete packet as it appears when it is sent out of the physical NIC port.

We describe the construction of a packet and its forwarding in terms of Linux structures and data flow. This process is all hardware offloaded, as described in Section 5, and maintains native RDMA hardware performance.

When an application sends traffic, the application knows the IP address of the destination (e.g., some other worker). The bridge forwarding database (③ in Figure 3) holds a mapping between IP addresses (H1_APP_IP) and MAC addresses (H1_APP_MAC). The bridge forwarding database (`br fdb`) resolves the destination MAC address and forms a RoCEv2 packet, as labeled in Figure 5.

The packet is then forwarded to the VTEP device. There, a lookup on the EVPN (Ethernet Virtual Private Network, ④ in Fig. 3) fdb table using the destination MAC address (H1_APP_MAC) provides information about the remote VTEP associated with that MAC address. This information includes the IP address of the host with the remote VTEP as well as the VNI of the VTEP. The application packet is then encapsulated (labeled as Tunnel in Figure 5). For

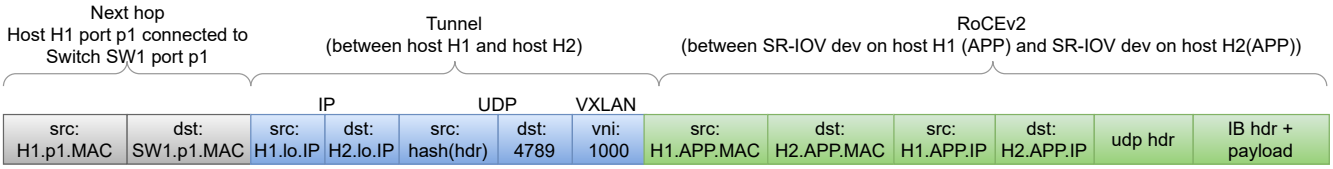

Figure 5: Structure of packets as they enter the network from a NIC.

VXLAN, this data includes an IP header (of the source and destination hosts, H1_LOOP_IP and H2_LOOP_IP), UDP header (which carries a fixed port for the destination), and VXLAN header (which carries the VNI).

Finally, the packet's destination IP address (which is the destination in the tunnel, or H2_LOOP_IP) is looked up in the routing table (⑤ in Fig. 3), which contains a list of next hops ordered by best selection. In this example, the packet is configured to go out on device p1. The source is filled in using the MAC address of device p1 and the destination is filled in using the MAC address of the port of the switch connected to device p1.

4.3 Scalable Path Selection with EVPN

The previous subsection discussed the various tables and processing steps for handling a packet. This subsection describes how THORN-ML populates and maintains those tables. THORN-ML uses standard routing software on each host, as shown in Figure 3. In particular, we use EVPN (Ethernet Virtual Private Network), a standard control plane using the BGP (Border Gateway Protocol) routing protocol. EVPN is used to enable two core functions of THORN-ML: first, it allows each router to advertise the VNIs of the VTEPs it is hosting. This approach provides an automated way for each router to know the IP addresses of all hosts with the same VNIs.

The second capability is what makes EVPN highly scalable, as it can be used to exchange the forwarding databases (FDBs). Typically, to determine the MAC address of a destination from an IP address, the Linux kernel issues an address resolution packet (ARP) request which forwards the message to all hosts. This approach is not scalable for larger networks. Instead, with EVPN, BGP is used to exchange this information.

BGP is highly scalable because it peers with neighboring routers. With this peer system, THORN-ML leverages BGP to share the forwarding database with neighboring nodes. At any point in time, any node knows the available network paths, as it learns about the routes available via the physical ports (dev p1 and dev p2) With this information shared, a node can select the best path and set the path into its own routing table (the last step discussed Section 4.2).

When failures occur at the network edge, they are detected in a fast manner through bi-directional forwarding detection (BFD). BFD is a protocol that can rapidly, and with low overhead, detect failures on a link between two connected devices [3, 17]. This protocol is integrated into the routing software. If a transceiver fails, a cable to the ToR fails, or the ToR itself fails, the routing tables in Linux will be updated quickly, thanks to the BGP peer system learning all possible paths and failure only requiring the removal of an entry. With the routing table in Linux updated, the last step of the packet processing previously discussed only sees one of the devices (between p1 and p2) as a possible next hop, and will choose that as a result.

5 Hardware Offload with Unmodified Hardware/Protocols

In Section 4, we detailed THORN-ML approach to provide a virtual device that can traverse different network ports (to different ToR switches) with standard Linux-based software. The system is designed such that it can be offloaded to hardware with minimal custom software.

5.1 Hardware offload with switchdev

With RDMA, local data (whether in system memory or GPU memory) should be directly accessible by the NIC, which can create the needed packet headers and transmit the packets. This means that the tables and configured transformations need to be offloaded to hardware. Without hardware offload, software in Linux needs to craft the packets, which is directly counter to one of the primary benefits of RDMA.

Linux provides a standard driver interface called *switchdev* [8] (⑥ in Fig. 3) which offloads the forwarding (data) plane from the kernel. Many NICs support this interface, including the Mellanox Connect-X 5 card used in our experiments.

One challenge is that the switchdev driver does not offload every configuration in Linux – e.g., it does not offload the VTEP device which performs encapsulation in a VXLAN header. However, one aspect that is well supported for offload are rules inserted with the traffic control (tc) utility.

The *tc* utility is quite powerful at expressing rules following the structure of a match-action pairing, with a rich set of match capabilities and a rich set of action capabilities. To understand, illustrated in Figure 6 is a general view of the match-action style processing. Header fields can be extracted to form a key, which is then used in a lookup table. That table identifies which action (or set of actions) should be executed based on the values of the keys. These actions can modify packets, encapsulate/decapsulate headers, forward packets, etc.

In the case of THORN-ML, the packet starts with an application destination IP and MAC address, and with this information the NIC determines the associated remote VTEP (IP address of the remote VTEP). Then, the NIC encapsulates the packet in a tunnel header, and forwards it based on the routing table. To achieve this processing, THORN-ML uses one type of match rule and two types of actions (⑦ in Fig. 3 and details shown in Fig. 7).

tc match: The destination APP MAC address is used to match with one rule per destination SR-IOV device (what the application

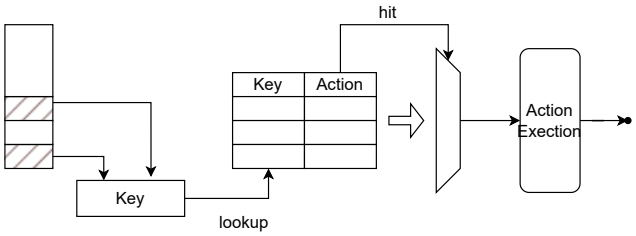

Figure 6: Match Action Table abstraction.

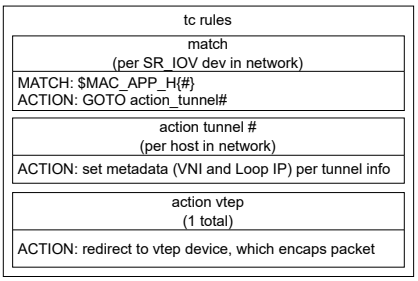

Figure 7: Traffic control rules used by THORN-ML.

binds to). Actions can be referenced by an index, allowing multiple matches to share the same action by having each match refer to the action index. This helps with scalability of THORN-ML, as each host may have multiple SR-IOV devices, and each corresponding match condition can map to the same action index.

tc action tunnel #: This action sets metadata about the tunnel, including the VNI and the LOOP_IP of the destination hosts' loopback device, using the tunnel_key action. One of these actions is required per destination host.

tc action vtep: This action redirects the packet to the VTEP device, which uses the metadata to encapsulate the packet in a VXLAN header, and pass the packet to be forwarded. Because we only use metadata, only one of these actions is needed per host.

The routing table used for forwarding is automatically offloaded by the switchdev system. This is especially useful since upon failure, the tc matches and actions described above do not need to change, only the routing table which maps destination IP address (i.e., H1_LOOP_IP) to a next hop must change. The change to the routing table is handled quickly by the Linux kernel driver.

5.2 Correctness despite a best effort protocol

In order for THORN-ML to translate between the kernel's network configuration that can not be offloaded and the offloadable tc rules needed to process packets, we include a Reconciler loop (⑧ in Fig. 3), a user-space application. The Reconciler loop uses Netlink [15], which is a socket family widely used for communication between a user space process and the kernel, to reconcile the two states. Netlink monitors the Linux kernel's network configuration and it can work in both a request-response mode (e.g., tell me the current Linux routing table), as well as a publish-subscribe mode (e.g., update me when the Linux routing table changes). A naïve implementation subscribes to the tables of interest, waits for an update, writes a new tc rule, and repeats. Unfortunately, Netlink

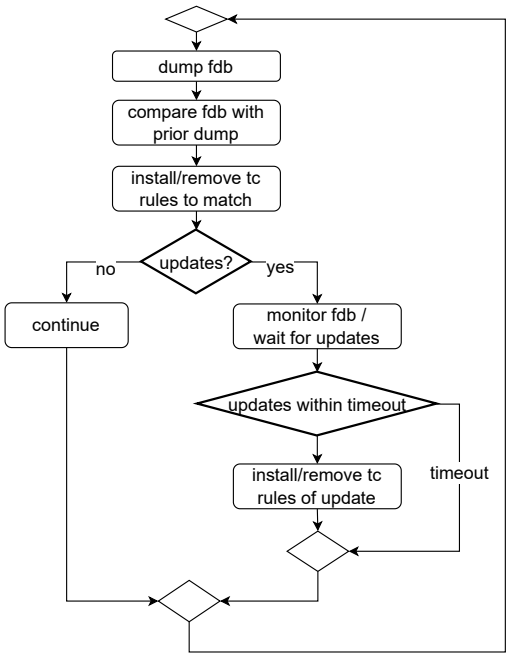

Figure 8: Reconciliation loop to ensure consistency between offload rules and Linux forwarding databases.

is a best-effort protocol that does not guarantee delivery [15]. For example, it may drop messages due to lack of network socket buffer space in the Linux kernel. To ensure correctness of THORN-ML Reconciler loop despite Netlink's best effort mechanism, we design an algorithm (the Reconciler Loop shown in Figure 8) that ensures convergence when reconciling between the network configuration state (as determined by the routing software) and the tc rules offloaded to hardware.

The Reconciler loop maintains a copy of its current view of Linux forwarding databases. This view is initialized by requesting a dump from the kernel (a full copy of the state). The Reconciler loop then blocks on updates from the kernel. When an update is received, the new state is recorded in the Reconciler's internal state. Updates are received by requesting a full dump of the internal state (which mitigates inconsistency due to incomplete updates from Netlink). In case of an update that is missed entirely, the Reconciler also periodically (based on a configurable timeout) requests a dump from the kernel. This algorithm is effective because it uses the update triggers from the kernel plus the periodic timeout to ensure a missed update is missed for no longer than the set timeout threshold, but avoids inundating the kernel with dump requests.

6 Evaluation

This section evaluates the effectiveness of THORN-ML, using unmodified applications on commodity hardware (Section 6.1). We evaluate THORN-ML on three core metrics: (i) impact on application due to failures, (ii) bandwidth, and (iii) latency. For the application progress we measure the degree to which failures in network edge components cause disruption in THORN-ML. We find that there is no noticeable impact on accuracy over time even with

frequent failures (Section 6.2). We also measure the impact on bandwidth of THORN-ML for collective communication operations. We find that there is negligible impact on the bandwidth, enabling applications to transfer data at the full rate of the underlying links. (Section 6.3). Finally we evaluate THORN-ML impact on RDMA transfer latency, during normal operation and during failure. We find that only packets that need to be re-transmitted incur additional latency, and that the detection and fail-over time is so small that it only shows in the 99.9th percentile. (Section 6.4)

6.1 Experimental Setup

6.1.1 Prototype and Testbed. The testbed where we run the THORN-ML prototype is a cluster of 5 hosts and 2 switches. Each host consists of an Intel Xeon Silver 2.10GHz 64-core CPU, an Nvidia A100 GPU with 16GB of memory, and a dual port, 100Gbps Mellanox ConnectX-5 network card (which supports RDMA). Each switch is a 48-port, 100Gbps Mellanox SN2700. The ports of each ConnectX-5 card are connected to different network switches, and the switches are connected to each other using an active-backup link pair. A 1TB storage node is connected to each network switch, providing a high-throughput, low-latency log collection and dataset distribution system.

THORN-ML uses Linux 6.1.87, FRR version 10.1 for routing software, a patched version of the rust-netlink library [21] (with the modifications being upstreamed), and the Mellanox OFED driver, version 24.01-0.3.3.1. To set up the eswitch on the NIC, we use the Mellanox Firmware Tool (mstflint) [14] along with standard Linux utilities (such as iproute2), totaling 467 lines of scripting code. Setting up the VTEPs are standard Linux utilities with 102 lines of scripting code. The Reconciler is written with 3679 lines of Rust. We will open source the prototype upon publication.

The testbed consists of only commodity components that are used in datacenters. This setup demonstrates that THORN-ML does not require any changes to hardware or protocols, and is highly deployable and interoperable.

6.1.2 Applications. We evaluate THORN-ML using three real unmodified applications: ResNet101, CodeParrot GPT-2 and NVIDIA NCCL. Evaluating with real, unmodified applications allows us to evaluate various modes of parallelism and communication patterns, while also demonstrating that THORN-ML is completely transparent to applications.

ResNet101

The Residual Network (ResNet) is a convolutional neural network designed to enable deep models: ResNet101 [26] is 101 layers deep. We train on the CIFAR10 [4] dataset, a collection of 60,000 32x32 color images that are categorized into 10 classes. We use a data parallel architecture, using the Pytorch Distributed Data Parallel (DDP) API.

CodeParrot GPT-2

To evaluate a complete, large scale application, where we can vary the parallelism architecture for evaluation, we use CodeParrot. CodeParrot [5] is a GPT-2-based language model specifically trained to enable users to generate Python code (in the style of GitHub CoPilot [10]). The authors behind CodeParrot provide datasets of code from GitHub [6] that allows researchers to train the model from scratch.

Table 2: Arguments for GPT-2 Training in Megatron-LM

Model Arch.	num-layers	12
	hidden-size	768
	num-attention-heads	12
Input/Output	seq-length	1024
	max-position-embeddings	1024
Training Params	micro-batch-size	12
	global-batch-size	192
	learning-rate	0.0005
	adam-beta2	0.999
HW/Precision	format	fp16

We conduct the experiments using the Megatron-LM framework [45], which is a research-oriented framework designed for training large-scale language models. It leverages Megatron-Core, a GPU-optimized library, and offers a flexible API for developing custom transformer models. Popular frameworks like HuggingFace Accelerate [2] are built on top of Megatron-LM. Table 2 lists arguments used for GPT-2 training in Megatron-LM. We run the GPT-2 training experiments on the CodeParrot dataset across three different architectures:

(1) **Pipeline (model) parallel:** The GPT-2 model consists of 12 layers. We use pipeline parallelism where the boundaries between workers is between layers. In this configuration, we divide the model into four parts to be distributed across four workers, each comprising of three layers.
(2) **Pipeline and tensor (model) parallel:** We use a hybrid parallel architecture, combining tensor and pipeline parallelism. The model is distributed across two nodes/GPUs horizontally and vertically, requiring a total of four GPUs/nodes for full model-parallel execution.
(3) **Data Parallel:** In this setting, each GPU maintains a complete copy of the model, while data is distributed in batches across the worker nodes.

NVIDIA Collective Communications Library (NCCL)

The NVIDIA Collective Communications Library (NCCL) [39] is a specialized library for high-performance GPU communication. NCCL offers a comprehensive set of collective operations, including all-reduce, all-gather, reduce, broadcast, reduce-scatter, all-to-all, and point-to-point communication. We test NCCL performance using NCCL Tests [39], a benchmarking tool designed to measure the performance of collective operations across multiple GPUs. NCCL Tests provides insights into the bandwidth achieved during the execution of these operations. Importantly, this setup provides a seamless mechanism for us to experiment with varying communication patterns in different configurations using a library used by many applications.

6.2 Impact on Applications Due to Failure

Core to THORN-ML is that a failure of a data center network edge component should not disrupt DDL applications (as current, application-level failure handling techniques do). We measure the actual impact of failures by inspecting the outcome of the application over time.

6.2.1 Accuracy over Time. In this experiment we measure the accuracy in ResNet101 training on CIFAR10 dataset using Pytorch DDP API for 150 epochs. We compare with a baseline of training under normal (non-fault) conditions (without any THORN-MLcomponents) versus one in which network faults are injected (with THORN-ML). This experiment shows THORN-ML ability to handle detection and re-routing without impacting accuracy.

Faults are injected based on a constant distribution. A fault cycle has a 40 second period, where the device is brought down for 20 seconds, brought back up for 20 seconds, and then the cycle is repeated. We repeat this cycle during the entire training time for both ResNet101 and GPT-2, which means that many faults occur.

We inject the faults at the network switch, rather than the host, to ensure there is no signal from the operating system of the port going down, and that the failure detection mechanism of THORN-ML is detecting the failure. We select the port on the device to bring down in a round-robin manner, In this experiment there is no noticeable difference in accuracy as a function of time (shown in Figure 9), indicating THORN-ML is able to keep the overall training job progressing – even in this extreme fault scenario.

We also compare against a theoretical training using checkpointing. The dotted line labeled "Single Fault + Checkpointing" represents a best case if there were only a single fault in the experiment that was subsequently fixed before restarting the training job. This experiment does not include the time to detect and initialize the workers from the checkpoint for the training job. Between 1 and N epochs of training get lost due to the failure, and Figure 9) illustrates the accuracy being stuck at the same amount until the training job re-starts (between 500s and 2500s). We assume checkpoints happen every 10 epochs, thus the accuracy gets stuck at the same amount for roughly 2000 seconds. If multiple faults occur during the training job, as it is the case for the experiment with THORN-ML, the restart delay occurs for every fault. The solid orange line labeled "Many Faults + Checkpointing", represents this case under the extreme fault scenario, in which we stressed the system injecting many faults. Traditional checkpointing alone is not enough to be able to make training progress since there is a fault in every epoch (whereas, THORN-ML is able to continue with similar progress as the baseline without faults).

6.2.2 Latency: Across Varying Parallelism Architectures. At a macro level, we do not see a discernible difference in accuracy over time between the case with faults using THORN-ML, and without any faults. We can investigate further by looking at the epoch latency.

This experiment uses the CodeParrot GPT-2 model, which allows us to change the parallelism architecture with simple configuration changes. Our evaluation includes configuration with pipeline, pipeline/tensor, and data parallelism architectures in Megatron.

We measure the time it takes to complete a training iteration. This metric takes into account the failure detection time, failure recovery time, and the associated drop in bandwidth due to the NIC re-transmitting packets as part of the reliably connected RDMA channel. Faults are again injected in the same manner as described in Section 6.2.1 (20 seconds down, 20 second up).

Figure 10, shows the cumulative distribution function (CDF) of the time to complete each iteration for the three architectures (data parallel, pipeline parallel, and pipeline and tensor parallel).

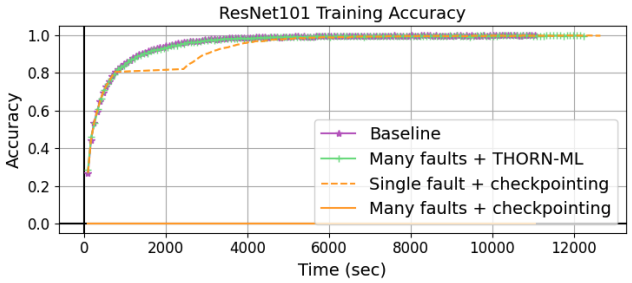

Figure 9: Accuracy of ResNet101 training on CIFAR-10 dataset using Pytorch DDP API on 4 nodes. Baseline is training without any faults without THORN-ML. Many faults is the test with a fault injected every 20 seconds and the link being down for 20 seconds, then repeat. Single fault refers to injecting a fault on a link, then recovering, and not injecting any more faults The line labeled with + THORN-MLis running THORN-ML, whereas the lines labeled with +checkpointing do not have any THORN-ML components running.

In all three configurations, we see little to modest impact. For data parallel, the added latency at the 90th and 95th percentile is 5.3ms and 7.3ms, respectively, representing a 1.1-1.6% overhead. For the pipeline parallel architecture, the 90th and 95th percentile have an additional 54.9ms and 76.7ms latency, for an overhead of 5.6-7.8%. In the combined pipeline and tensor parallel architecture, the 90th and 95th percentile have an additional 729.4ms and 928.4ms of added latency, representing a 9.2-11.7% overhead.

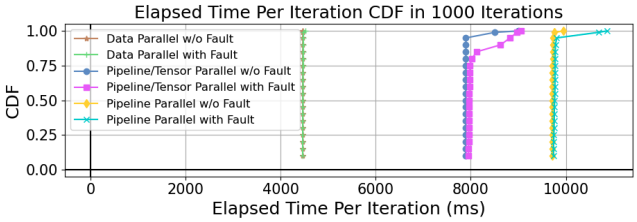

Figure 10: Distributed GPT-2 training on CodeParrot dataset. Each data point is averaged over 10 iterations. Lines labeled 'w/o fault' refer to no components of THORN-ML running, and no faults are injected during the life of the experiment. Lines labeled 'with fault' refer to when THORN-ML is running and faults are injected during the experiment in a repeated cycle of 20 seconds down then 20 second up.

6.3 Bandwidth: Varying Communication Operations

Next, we explore the impact on bandwidth for different communication operations. We use the NCCL test benchmark suite, which provides the means to test several different collective communication operations, while also varying the message size. We run tests with 100MB, 250MB, 1GB, and 2GB message sizes, all with the same amount of data exchanged ($\approx 32GB$ in total).

Figure 11 is the graph for 2GB message size, which is representative of the results for all message sizes. For example, at the 80th

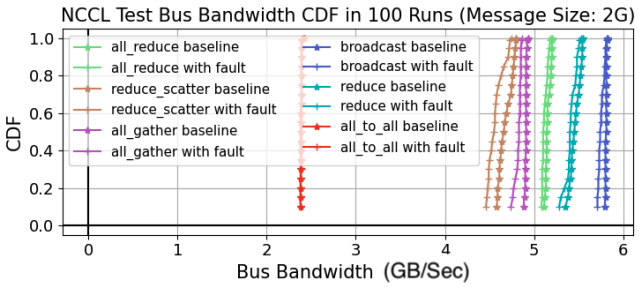

Figure 11: NCCL test collective operations performance report with a message size of 2GB. Lines labeled 'baseline' refer to a case where no components of THORN-MLare running, and no faults injected. Lines labeled 'with fault' have THORN-ML and a single fault is injected and the fault persists for the remainder of the experiment.

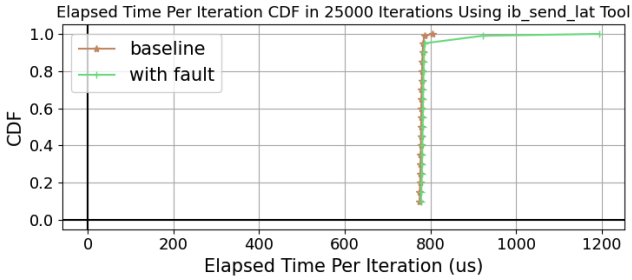

Figure 12: Latency CDF using RDMA performance test

percentile, across all operations there is a maximum difference of 0.129Gbps between the fault and non-fault scenarios. One can note that the throughput achieved is not maxing out the line rate of the system. We believe this to be a system configuration issue, but the takeaway remains that there is minimal overhead – both the baseline (no THORN-ML components, and no faults injected) and with fault (with THORN-ML running and faults injected) are subject to the same system configuration and exhibit very close throughput.

Based on the link speed of 100Gbps, these tests only show a maximum of 6GB/sec. We attribute this difference to the known challenge of temporary bandwidth drops upon packet loss in RDMA, which occur in the case when the failure is unplanned and, as such, causes packet losses.

6.4 Failure Recovery Time

While previous experiments illustrate the impact to the application performance, here we look into system performance at a finer granularity by measuring the latency impact of a single fault. For this, we leverage an RDMA performance test benchmark tool.

To assess the impact of network failures on communication latency, we conduct a series of send transaction experiments between two nodes. Using the *ib_send_lat* command with a message size of 2^{23} bytes, we measure the latency distribution for both normal conditions and scenarios with injected faults occurring midway through the experiment. The resulting graph, shown in Figure 12, depicts the cumulative distribution function (CDF) of iteration latency throughout the entire experiment. Note that data points with higher delay are due to re-transmission, and the number of packets that need to be re-transmitted indicates the fail-over time. We see that only packets that need to be re-transmitted incur added latency, and that the detection and fail-over time is so small that it only applies in the 99.9th percentile.

7 Related Work

THORN-ML addresses the problem of resilience in DDL applications. There are two main bodies that we see as most related.

Application Level Fault Handling: The core approach that distributed deep learning frameworks use are around checkpointing [22, 23, 28]. HopLite is one of these frameworks [52], which implements collective communication operations atop task-based systems that allow workers to join and leave dynamically. While this system improves resilience, network failures at the edge are still disruptive even. THORN-ML is a complimentary approach that can be used in combination with HopLite to provide more comprehensive resilience. Ultima [51] is another technique that leverages adaptive timeouts and transpose-all reduce collectives to reduce the impact of dropped or missed gradients. However, its target is to reduce nominal packet loss (e.g., due to congestion, not failure), and it is unclear how it would work under full failure scenarios (especially a top-of-rack switch failure).

Within an application are the use of libraries. In this approach, closest to THORN-ML is LubeRDMA [36], which introduces a new library that provides a shadow copy of the control resources in the RDMA context, with the ability to change the data path. LubeRDMA, however, increases latency per their evaluation, has overhead of setting up a new connection upon failure, only posits that it would work with GPU Direct, and is not transparent to the application (as it needs to use a different library, or incorporate this functionality into a variety of existing libraries [11, 13, 20, 39, 52]). A key contribution of our work is that we evaluate the impact of failures with and without THORN-ML on distributed deep learning. LubeRDMA does not demonstrate with any applications.

Multi-path RDMA: Recent work by Google reported that the static routing decisions used with RoCE in large scale systems (in this case 4096 nodes) can cause a slowdown of up to 9% and for all-reduce workloads the throughput can be degraded by up to 50% [53]. Meta reported that training jobs had a degradation of more than 30% due to network fragmented traffic and uneven network distributions [32]. These motivated a long line of work in multi-path RDMA, to address the deficiencies of equal cost multipath (ECMP). For example, *Maestro*, leverages software middleware to extend the RDMA library in Pytorch [48], though it is not transparent to applications like THORN-ML. *ConWeave*, leverages Tofino2 switches to address congestion and out-of-order packets [46] in the core of the network, so it is complimentary to THORN-ML.

The closest work to THORN-ML, which proposes a custom FPGA NIC and extensions to the RDMA protocol to do network load balancing is by Lu *et al.* [37]. This prior work is not transparent to the application and does not work with commodity hardware. Others, either require significant modifications to the applications (whereas THORN-ML is completely transparent) or target the core of the network (which is complementary to THORN-ML).

8 Discussion

Scalability: We tested on a 5-node cluster due to resource constraints, but understand that this falls short of the cluster sizes that motivate the design of THORN-ML. We expect THORN-ML to scale to well beyond 100,000 nodes based on the following three dimensions: protocol, memory and latency.

First, scalability of THORN-ML is dependent on the limitations of the protocols used. For the data plane, THORN-ML leverages VXLAN where we use the same VNI for all hosts in the cluster, and we use the UDP destination field to address the individual tunnel endpoints. As such, THORN-ML is currently limited by the number of bits in the UDP destination field (32). This limits a cluster to include up to 2^{32} hosts (4 billion), which is well beyond what is needed. For the control plane, THORN-ML uses BGP for communicating new routing paths. The amount of traffic, even in a large failure scenario of a ToR switch, would be kilobits of traffic.

Second, THORN-ML scalability depends on the limited memory on network cards to store the *tc* rules. As discussed in Section 5, THORN-ML has one *tc* rule per host. Assuming an aspirational cluster size of 100,000 nodes, for example, THORN-ML would only require 100s of kilobytes to store the rules in the SmartNIC memory. Azure reported their SmartNIC (7 years ago) handles 1M rules [31], which would suggest a scale of around 1M nodes for THORN-ML.

Finally, the latency to update the rules may play a role in the scalability of THORN-ML. To quantify, we simulated a ToR failure event with paths to 10s of nodes needing to be updated (i.e., those connected to that ToR). The reconciliation loop and the rule insertion into Linux were on the order of a couple milliseconds.

Multiple NICs: THORN-ML is addressing the unsolved problem of seamlessly supporting multiple ports on a NIC, so there remains a question of how THORN-ML would work with multiple NICs. We view this as an orthogonal problem, and one for which there are production solutions.

Consider the Dell Rail-optimized network architecture, designed to support the largest of distributed training workloads [1]. To support a large scale system with many servers and multiple GPUs per server, Dell designed an architecture where each GPU gets associated with a dedicated RDMA capable network adapter (forming different rails) – enabling communication with GPUs in other servers. For cross-rail communication, they note that PXN (PCI × NVLink), introduced in NCCL 2.12, allows a GPU to communicate with a NIC on the node via NVLink and then PCI.

What the Dell rail-optimized network architecture (or any other architecture, for that matter) cannot do is support multiple ports per NIC. That is, it cannot handle detection of a port failure and rapidly reconfigure the NIC's eSwitch to change paths. We believe these two technologies will work seamlessly together, and plan to validate this claim as future work.

Isolation / Security: In this work, we presented the design, implementation, and evaluation with a single tenant context – a company performing some large training job. However, cloud environments are multi-tenant and even private training clusters likely run multiple training jobs simultaneously (each job can be considered a tenant). THORN-ML is designed with multi-tenancy requirements in mind, specifically isolation and security.

At a foundational level, each tenant gets a collection of Linux devices (an SR-IOV device, a VTEP device, etc.) created. The virtual device that is exposed to a tenant in a container or VM environment, is the device that the tenant is able to configure. This setup is part of the cluster / cloud management software, which authenticates a user and provisions system resources.

Isolation between tenants is important to ensure they cannot interfere with one another. Given that each tenant is using a different number of rules (offloaded to hardware, using limited space), and installing at different rates, some interference may be present. We do not believe this interference is something a tenant can directly impact, but it needs further investigation. Regardless of this limitation, this problem can be solved through an update to the switchdev driver as future work.

Overhead: THORN-ML introduces additional software to each host, as well as tunnel traffic encapsulation. Each adds a degree of overhead. For the dataplane, VXLAN headers are 8 bytes long. DNN training jobs have large packets due to the amount of data communicated. Standard ethernet frames are 1500 bytes, representing a 0.5% overhead, and jumbo frames are 9000 bytes, representing a 0.08% overhead on the network traffic.

For the control plane, as previously discussed, there is minimal traffic generated by BGP, even in failure cases. The software on the host requires memory to store the routing tables, which is on the order of 10 megabytes at most. THORN-ML requires minimal CPU, except to process updates in failure cases, but it does not interfere with the training jobs which leverage GPUs.

9 Conclusions and Future Work

This paper presents THORN-ML, an architecture for resilience at the network edge that requires no modification to distributed deep learning applications and can work on commodity hardware and existing popular protocols (RDMA). For application transparency, we introduce a virtual network device that applications can bind to coupled with routing software on the end-hosts to detect failure and select the best routing paths. THORN-MLworks with commodity hardware through the support of the switchdev driver, where we bridge the gap between the routing software and the offload capabilities through a reconciliation loop performing rule translation. The end result is a system that enables real applications to proceed without needing to revert to a checkpointed state when encountering a failure at the edge of the network and thus reducing disruption from minutes to milliseconds. Among possible future directions, we will explore integration with the complimentary multi-path RDMA solutions. We will also look to find partners that would enable us to test THORN-ML at larger scales, including testing systems with multiple NICs and GPUs.

Acknowledgements

This work was supported in part by the National Science Foundation (NSF) grant number 2241818. Giulio Sidoretti completed this work while a visiting researcher at the University of Colorado, Boulder under the NGI Enrichers program. Some of the code originated, with permission, from efforts at Stateless, Inc. which both Daniel Noland and Eric Keller had a past affiliation.

References

[1] 2024. Dell Technical White Paper (H20003)–Generative AI in the Enterprise – Model Training (Network Architecture section). https://infohub.delltechnologies.com/pt-br/l/technical-white-paper-generative-ai-in-the-enterprise-model-training/network-architecture-155/.

[2] 2025. Accelerate. https://huggingface.co/docs/accelerate/en/index

[3] 2025. Bidirectional Forwarding Detection Commands on the Cisco IOS XR Software. https://www.cisco.com/c/en/us/td/docs/routers/xr12000/software/xr12k_r4-1/interfaces/command/reference/interfaces_cr41xr12k_chapter2.html

[4] 2025. cifar10 | TensorFlow Datasets. https://www.tensorflow.org/datasets/catalog/cifar10

[5] 2025. codeparrot/codeparrot - Hugging Face. https://huggingface.co/codeparrot/codeparrot

[6] 2025. codeparrot/codeparrot-clear - Datasets at Hugging Face. https://huggingface.co/datasets/codeparrot/codeparrot-clean

[7] 2025. CoreWeave. https://www.coreweave.com/.

[8] 2025. Ethernet switch device driver model (switchdev) – The Linux Kernel documentation. https://docs.kernel.org/networking/switchdev.html

[9] 2025. FRRouting. https://frrouting.org/

[10] 2025. GitHub CoPilot – Your AI pair programmer. https://github.com/features/copilot

[11] 2025. Gloo. https://github.com/facebookincubator/gloo.

[12] 2025. GoBGP. https://osrg.github.io/gobgp/

[13] 2025. Intel oneAPI Collective Communications Library. https://www.intel.com/content/www/us/en/developer/tools/oneapi/oneccl.html.

[14] 2025. MSTFLINT Package - Firmware Burning and Diagnostics Tools. https://github.com/Mellanox/mstflint

[15] 2025. netlink(7) — Linux manual page. https://man7.org/linux/man-pages/man7/netlink.7.html

[16] 2025. Network Function Representors – The Linux Kernel documentation. https://docs.kernel.org/networking/representors.html

[17] 2025. Network Lessons: Bidirectional Forwarding Detection (BFD). https://networklessons.com/cisco/ccie-routing-switching/bidirectional-forwarding-detection-bfd

[18] 2025. NVIDIA GPUDirect. https://developer.nvidia.com/gpudirect.

[19] 2025. Ray Train. https://docs.ray.io/en/latest/train/train.html

[20] 2025. RCCL documentation. https://rocm.docs.amd.com/projects/rccl/en/latest/.

[21] 2025. rust-netlink. https://github.com/rust-netlink

[22] 2025. Saving and loading a general checkpoint in PyTorch. https://pytorch.org/tutorials/recipes/recipes/saving_and_loading_a_general_checkpoint.html

[23] 2025. Saving and Loading Checkpoints – Ray 2.38.0. "https://docs.ray.io/en/latest/train/user-guides/checkpoints.html"

[24] 2025. Single Root IO Virtualization (SR-IOV) – NVIDIA Docs. https://docs.nvidia.com/doca/sdk/single+root+io+virtualization+(sr-iov)/index.html

[25] 2025. TensorFlow v2.16.1 API: MinitoredTrainingSession. https://www.tensorflow.org/api_docs/python/tf/compat/v1/train/MonitoredTrainingSession.

[26] 2025. tf.keras.applications.ResNet101 | TensorFlow v2.16.1. https://www.tensorflow.org/api_docs/python/tf/keras/applications/ResNet101

[27] 2025. The BIRD Internet Routing Daemon. https://bird.network.cz/

[28] 2025. Training Checkpoints | TensorFlow core. https://www.tensorflow.org/guide/checkpoint

[29] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[30] Dinesh G Dutt. 2018. *EVPN in the Data Center*. O'Reilly Media, Inc.

[31] Daniel Firestone et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[32] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. 2024. RDMA over Ethernet for Distributed Training at Meta Scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*. 57–70.

[33] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*.

[34] Leslie Lamport and K. Mani Chandy. 1985. Distributed Snapshots: Determining Global States of a Distributed System. *ACM Transactions on Computer Systems* 3, 1 (Feb. 1985).

[35] Kevin Lee, Adi Gangidi, and Mathew Oldham. 2024. Building Meta's GenAI Infrastructure. https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/.

[36] Shengkai Lin, Qinwei Yang, Zengyin Yang, Yuchuan Wang, and Shizhen Zhao. 2024. LubeRDMA: A Fail-safe Mechanism of RDMA. In *Proceedings of the 8th Asia-Pacific Workshop on Networking* (Sydney, Australia) *(APNet '24)*. Association for Computing Machinery, New York, NY, USA, 16–22. https://doi.org/10.1145/3663408.3663411

[37] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. 2018. {Multi-Path} transport for {RDMA} in datacenters. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*. 357–371.

[38] Mallik Mahalingam, Dinesh G. Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T. Sridhar, Mike Bursell, and Chris Wright. 2014. RFC 7348: Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. https://datatracker.ietf.org/doc/html/rfc7348

[39] NVIDIA. 2025. NCCL. https://github.com/NVIDIA/nccl

[40] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. 2022. Jupiter Evolving: Transforming Google's Datacenter Network via Optical Circuit Switches and Software-Defined Networking. In *Proceedings of ACM SIGCOMM 2022*.

[41] Pytorch. 2025. Pytorch Distributed Data Parallelism. https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

[42] Robi Rahman, David Owen, and Josh You. 2024. Tracking Large-Scale AI Models. https://epochai.org/blog/tracking-large-scale-ai-models.

[43] Renato J. Recio, Bernard Metzler, Paul R. Culley, Jeff Hilland, and Dave Garcia. 2007. *A Remote Direct Memory Access Protocol Specification*. Technical Report RFC 5040. Internet Engineering Task Force.

[44] Hemal Shah, Felix Marti, Wael Noureddine, Asgeir Eiriksson, and Robert Sharp. 2014. *Remote Direct Memory Access RDMA Protocol Extensions*. Technical Report RFC 7306. ISSN: 2070-1721. Internet Engineering Task Force.

[45] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *CoRR* abs/1909.08053 (2019). arXiv:1909.08053 http://arxiv.org/abs/1909.08053

[46] Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, and Mun Choon Chan. 2023. Network load balancing with in-network reordering support for rdma. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 816–831.

[47] Tensorflow. 2025. Tensorflow Multiworker Mirrored Strategy. https://www.tensorflow.org/api_docs/python/tf/distribute/MultiWorkerMirroredStrategy

[48] Feng Tian, Yang Zhang, Wei Ye, Cheng Jin, Ziyan Wu, and Zhi-Li Zhang. 2021. Accelerating distributed deep learning using multi-path RDMA in data center networks. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*. 88–100.

[49] Saranyan A. Vigraham and Benjamin Leonhardi. 2024. Maintaining large-scale AI capacity at Meta. https://engineering.fb.com/2024/06/12/production-engineering/maintaining-large-scale-ai-capacity-meta/.

[50] Pablo Villalobos and Anson Ho. 2022. Trends in Training Dataset Sizes. https://epochai.org/blog/trends-in-training-dataset-sizes.

[51] Ertza Warraich, Omer Shabtai, Khalid Manaa, Shay Vargaftik, Yonatan Piasetzky, Matty Kadosh, Lalith Suresh, and Muhammad Shahbaz. 2023. Ultima: Robust and Tail-Optimal AllReduce for Distributed Deep Learning in the Cloud. arXiv:2310.06993 [cs.DC] https://arxiv.org/abs/2310.06993

[52] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. 2021. Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 641–656. https://doi.org/10.1145/3452296.3472897

[53] Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, et al. 2024. Resiliency at Scale: Managing {Google's}{TPUv4} Machine Learning Supercomputer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 761–774.