# Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer

Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, and Henri Bahini, *Google*

https://www.usenix.org/conference/nsdi24/presentation/zu

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

# Resiliency at Scale: Managing Google's TPUv4 Machine Learning Supercomputer

Yazhou Zu, Alireza Ghaffarkhah, Hoang-Vu Dang, Brian Towles, Steven Hand, Safeen Huda, Adekunle Bello, Alexander Kolbasov, Arash Rezaei, Dayou Du, Steve Lacy, Hang Wang, Aaron Wisner, Chris Lewis, Henri Bahini
*Google*
*tpuv4-nsdi-paper@google.com*

## Abstract

TPUv4 (Tensor Processing Unit) is Google's 3rd generation accelerator for machine learning training, deployed as a 4096-node supercomputer with a custom 3D torus interconnect. In this paper, we describe our experience designing and operating the software infrastructure that allows TPUv4 supercomputers to operate at scale, including features for automatic fault resiliency and hardware recovery. We adopt a software-defined networking (SDN) approach to manage TPUv4's high-bandwidth inter-chip interconnect (ICI) fabric, using optical circuit switching to dynamically configure routes to work around machine, chip and link failures. Our infrastructure detects failures and automatically triggers reconfiguration to minimize disruption to running workloads, as well as initiating remediation and repair workflows for the affected components. Similar techniques interface with maintenance and upgrade workflows for both hardware and software. Our dynamic reconfiguration approach allows our TPUv4 supercomputers to achieve 99.98% system availability, gracefully handling hardware outages experienced by ~1% of the training jobs.

## 1 Introduction

Machine Learning (ML) models continue to grow in size and complexity [9, 24], enabled by the massive compute capability of heterogeneous supercomputers, where CPUs handle runtime task coordination and I/O, and accelerators such as TPUs [18–20] and GPUs deliver the computational performance needed for model training. Scaling up a supercomputer's node count enables more capable models because the training process can be effectively parallelized along the batch, tensor, and pipeline dimensions [17, 33].

The hardware/software ecosystem of ML supercomputers faces two challenges at scale: first, effectively parallelizing model training workloads, and secondly – and the focus of this paper – maintaining high availability of compute resources and consequently high goodput for ML training jobs. The latter has become increasingly difficult in recent years because:

- unlike loosely-coupled distributed applications such as Map-Reduce [12] that can effectively tolerate dynamically varying resource allocation, ML training jobs more often use static (compile-time) sharding strategies and gang scheduled execution, requiring all compute resources to be healthy simultaneously;
- modern ML models such as Large Language Models (LLMs) need an unprecedented amount of hardware (conventional compute, accelerators, networking and storage) [2], dropping the expected MTBF to hours or even minutes [15];
- in a cloud or shared cluster environment, many users contend for different subsets of supercomputer resources, making it essential to be able to reconfigure or rebalance resource allocations over time.

Google's TPUv4 machine learning supercomputing infrastructure is designed to meet these challenges. It comprises the following hardware and software components:

- A number of *cubes*: a cube is a hardware unit with 64 TPU chips arranged in a 4x4x4 3D mesh; each supercomputer or *pod* has 64 cubes, for a total of 4096 TPUs.
- A proprietary inter-chip interconnect (*ICI*): this is a high-speed network fabric that directly interconnects TPUs to allow direct device-to-device communication (i.e. RDMA) without involving the CPUs.
- Optical circuit switches (*OCSes*) [25]: these are used to dynamically cross-connect (xconnect) the ICI from different cubes to form the user-requested torus topology.
- Borg [31]: a cluster management service that admits, schedules and manages TPUv4 jobs (and others).
- Pod Manager: a cluster-level software service that manages multi-cube connectivity by actuating OCS xconnect setup in response to Borg scheduling decisions.

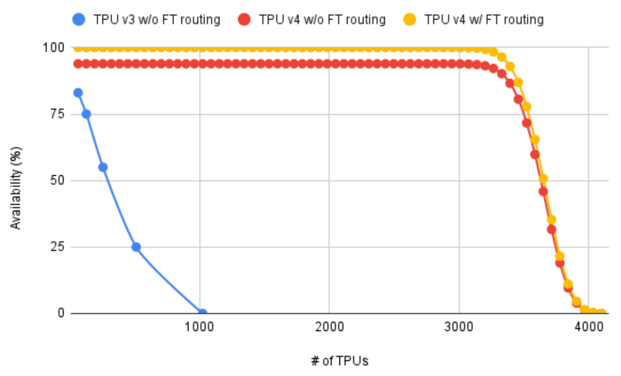


Figure 1: Availability of jobs size improves at scale with TPUv'4 cube configurability and fault-tolerant ICI routing.

- `libtpunet`: a software library that sets up the requested ICI network topology for each TPUv4 user job.
- `healthd`: a software daemon running on each host in a pod that continuously monitors machine hardware health and reports back to cluster-level software systems.

In TPUv4 pods, hardware and software are co-designed. Hardware provides a configurable compute substrate (the 4x4x4 cubes) with a programmable ICI protocol stack implemented by the OCS switches and on-chip ICI switches. Software dynamically manages the hardware by configuring the OCSes to combine multiple cubes into larger *slices* of the pod, and by programming the ICI routing policy via `libtpunet`. Connectivity is reconfigured according to the user-requested torus topology as Borg schedules jobs onto the various cubes. The Pod Manager mediates reconfiguration, and monitors ICI- and OCS-related health, excluding cubes from the resource pool as soon as faults are discovered.

In this paper, we describe how we make our TPUv4 supercomputers automatically resilient to faults at scale. More specifically, we:

– explain the TPUv4 supercomputer's configurable system architecture based on a programmable ICI protocol with OCS and ICI switches, optimizing for system availability and resiliency at scale;

– describe the software infrastructure that schedules, configures, and optimizes TPUv4 resources, focusing on our design principles of configurability and modularity;

– outline our optimized strategy for accelerator-side multi-hop RDMA routing for resilient collective operations over regular and twisted torus topologies; and

– report on our experiences to date in operating TPUv4 supercomputers in production.

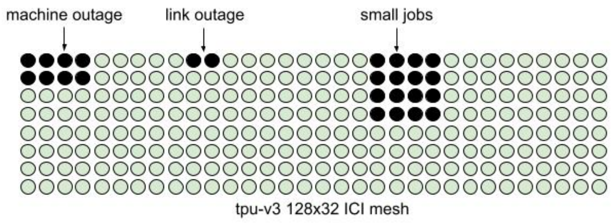


Figure 2: Static pods face resource fragmentation problems.

## 2 The Reconfigurable ML Supercomputer System Architecture

The TPUv4 reconfigurable supercomputer is designed for scalability, availability, resiliency, and cost [18]. At its core is a reconfigurable ICI fabric topology that connects different TPUv4 chips, backed by a set of programmable OCSes for each pod. Without TPUv4's OCS-based reconfigurability, job availability quickly drops as compute resource scales up. Figure 1 shows this effect with measured data from deployed TPUv3 *static pods* and TPUv4 *reconfigurable pods*.

For a conventional supercomputer like TPUv3 [19] where compute resources are statically interconnected, the overall availability of a job drops precipitously as the required amount of compute resources increases to 1024 chips. This is because in a static pod, all resources in a contiguous set of nodes must be simultaneously healthy to be assigned to a user, which becomes combinatorially less likely as the system scales. With TPUv4's cube-level configurability, availability stays high through about 94%, corresponding to ~50 cubes or 3200 TPUv4 chips.

The decreasing availability beyond this point is because of occasional machine and ICI link faults that can occur between different cubes. As we will show in Section 4, tolerating occasional OCS failures or maintenance events with fault-tolerant routing further increases availability to 99.98% because cubes are still accessible to users even in these rare events.

### 2.1 Lessons from Static Pod Architectures

Before the TPUv4 ML supercomputer, the state of the art was the TPUv2 and TPUv3 *static* pods [19] – 'static' because they feature a non-reconfigurable fixed ICI mesh. A TPUv2 pod has 256 TPUs connected in a 16x16 ICI torus, while TPVv3 had 1024 TPUs connected with a 32x32 torus. There is also a scale-up variant of TPUv3 that combines 4 pods into a 128x32 mesh with limited ICI routing capability. This so-called *multipod* version was co-designed with application collective patterns and used to explore scaling strategies for large models [21].

Figures 1 and 2 illustrate the availability challenges as the model size scales in a static pod. To train a model, all TPU processes must be simultaneously up to synchronously update their weights via ICI collectives. A single failed, or interrupted process will interrupt the whole training process.

Finding the appropriate compute resources for a user's job faces the following challenges:

1. Hardware outages: regular scheduled *maintenance* of hardware, firmware, and software at the ICI link, TPU chip and CPU host level can remove resources from the schedulable pool [22]. For a supercomputer with thousands of TPUs, an event affecting any one component occurs relatively frequently, making it difficult to find usable sets of resources. Furthermore, unexpected *faults* occur more frequently as systems and applications increase in both size and complexity. Without reconfigurability, obtaining decent availability for a job that requires 1024 hosts means that each individual host must sustain 99.9% availability; introducing reconfigurable OCS drops the host availability requirement to 99%.

2. Workload defragmentation: it is common for many jobs to contend for different subsets of a pod's schedulable resources. Since these jobs come and go at unpredictable times, sometimes Borg must move (preempt) smaller jobs to free up contiguous TPUs for pending larger training jobs. The scheduling complexity worsens with a mixture of user priorities. With OCS-based reconfigurability, Borg does not need to worry as much about the physical contiguity of TPU resources. Instead, any set of vacant cubes can be cross connected via the OCS for use by a user's job.

3. Deployment lead time: a static pod is not usable until *all* hardware is installed due to the tightly coupled nature of compute and network resources. With reconfigurable pods, once the OCS footprint is installed, cubes can be deployed and used as soon as they land.

The above challenges challenges motivated us to rethink things for the TPUv4 pod architecture.

## 2.2 TPUv4: OCS-based Reconfigurability

TPUv4 adopts a reconfigurable architecture which makes use of the Palomar Optical Circuit Switch (OCS) [25] to address the problems with static systems. By adopting this architecture, we have been able to effectively scale to 4096 TPU nodes, and to support a per-job choice of either 3D torus or 3D twisted-torus [7] topology.

The OCS is a dynamically configurable $N \times N$ switch based on an array of micro-electromechanical systems (MEMS) mirrors that can switch in milliseconds. Each OCS allows programmable cross-connect creation (xconnect) between any pair of ports on the (logical) north side of the switch to the (logical) south side. Once a connection between an $N_i$ to $S_j$ port is made, a dedicated ICI link connection is established such that optical signal from $N_i$ can only be routed to $S_j$ and vice versa, until these ports are reconfigured in some different permutation.

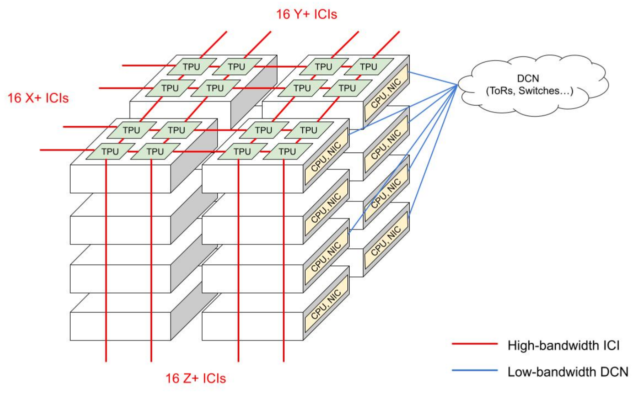


Figure 3: A 4x4x4 cube consists of 16 TPUv4 machines, each of which organizes 4 TPUs in a 2x2x1 mesh. The TPUs in a cube are interconnected over ICIs along $X/Y/Z$ dimensions, with 16 optical links per cube face for OCS xconnect.

TPUv4 compute resources are organized at the granularity of multi-machine *cubes*. Each individual TPU machine has a CPU tray and a TPU tray, linked over PCIe. Each TPU tray has 4 TPUv4 chips arranged in a 2x2x1 ICI mesh; 16 TPU machines are grouped together as one datacenter rack; and the ICI links within the rack are interconnected over ICI to form a 4x4x4 mesh. This ensemble is a cube.

The optical switches interconnect multiple cubes to form larger ICI topology shapes with one or more cubes in each of the three dimensions. Each 3D cube exposes 16 ICIs on each face of the $X/Y/Z$ dimension to the optical switches, totaling 96 ICIs per cube. A TPUv4 supercomputer consists of 64 cubes, with a total of 6144 optical ICI links connected to 48 distinct optical circuit switches. The lower-bandwidth CPU-side datacenter network is managed separately [25, 29].

TPUv4's OCS configurability greatly improves availability. Training jobs can use any cubes even if they are not physically contiguous, which mitigates resource fragmentation from competing jobs. Hardware failures remove the affected cube(s) from the resource pool but allow continued operation using healthy cubes. The 16-machine granularity for fault-tolerance was chosen to balance convenience (per rack deployment, power and networking) while retaining a relatively small blast radius in case of failure.

Reconfigurability is managed by the accompanying software infrastructure. Each job launch induces the software to establish a unique OCS xconnect depending on the required topology and cube selection. The large number of chips, links, and switches also requires automatic fault diagnosis, recovery, job rescheduling, and fault-tolerant ICI routing.

Using OCS scales TPUv4 pod with low cost: the OCS and optical fiber costs are $< 5\%$ of a TPUv4 pod's total capital cost, and their operating power is $< 3\%$ of a pod's total power. The capital and operating cost of TPUv4 OCS supercomputer is considerably lower than the alternative of scaling with packet switches such as Infiniband [18].

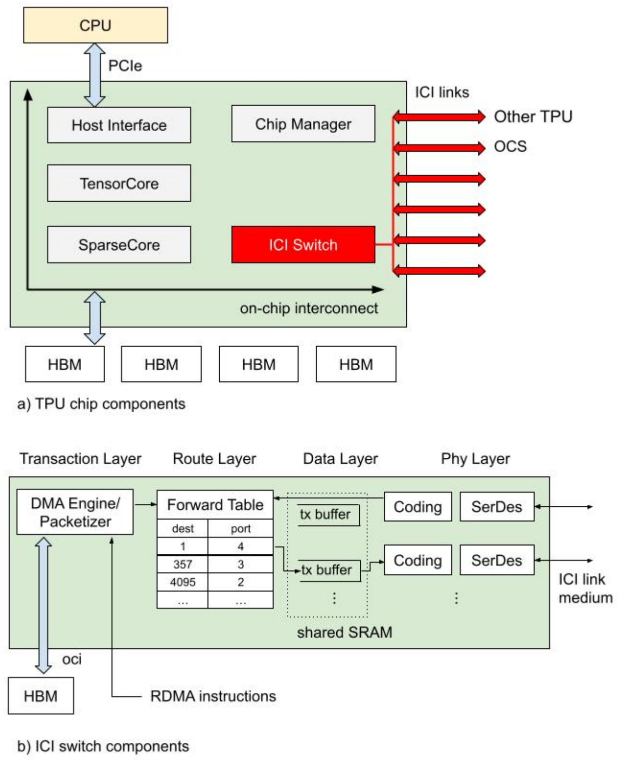


Figure 4: TPUv4's ICI switch implements layered, programmable ICI protocol.

## 2.3 Programmable ICI Protocol

TPUv4's ICI protocol is designed to be programmable so that software can tackle the operational complexity of reconfigurability and resilience. A TPUv4 pod is one ICI domain, where any pair of TPUs can RDMA to each other. Each ICI link can carry 50GBps uni-directional bandwidth. TPUv4 adopts a 3D ICI network topology for high bisection throughput, large system scale and low latency while maintaining low cost and supporting workload parallelization via collectives.

As shown in Figure 4, each TPUv4 chip has some compute, some high-bandwidth memory and an ICI switch that implements various ICI protocol layers. The ICI protocol facilitates per-job network partitioning, where connectivity, addressing, routing, and flow control are set up for each job, and where user sessions do not cross job boundaries. In this way, each job has exclusive ownership of all the links it uses, increasing security and removing additional system complexity for network sharing and congestion control. Table 1 shows the protocol layers and their corresponding software agents. From the bottom up, these are:

- **Physical Layer**: the SERDES, PCS, and link auto-establishment modules build a high-speed link, despite the inevitable presence of transmission errors. The Pod Manager controls xconnect of a physical channel by rotating OCS MEMS mirrors, and an on-chip manager automatically initializes and configures the physical links. The `healthd` daemon running in every TPU machine's Linux system container continuously reads link quality and connectivity signals to track hardware health.

| Layer | Functionality | S/W Agent | ISA Visible? |
|---|---|---|---|
| Transaction | RDMA | XLA | **yes** |
| Routing | packet forwarding | `libtpunet` | *hint* |
| Data | link enable, flow control, retry, ordered delivery | `libtpunet` `healthd` | no |
| Physical | link xconnect port training | pod mgr, chip mgr, `healthd` | no |

Table 1: ICI protocol layers.

- **Reliable Data Layer**: Packets are delivered in-order with automatic retransmit when data is lost at the physical layer, thus hiding the unreliable characteristics of the physical layer. Link-level, credit-based flow control is enforced. An *enabled* data layer signifies a ready-to-use ICI user session; prior to becoming ready, the system clears all data buffers to ensure we eliminate any architectural state pollution from prior ICI sessions. If one end of an enabled data layer is down, we automatically bring down the other end of the link to ensure a functional session. The `libtpunet` issues session start/stop commands, and adjusts optimal flow control buffer sizes. The privileged `healthd` machine daemon can explicitly disable a data layer link, forbidding its usage by any user session, in the case of an online link recovery (§3.6.3).

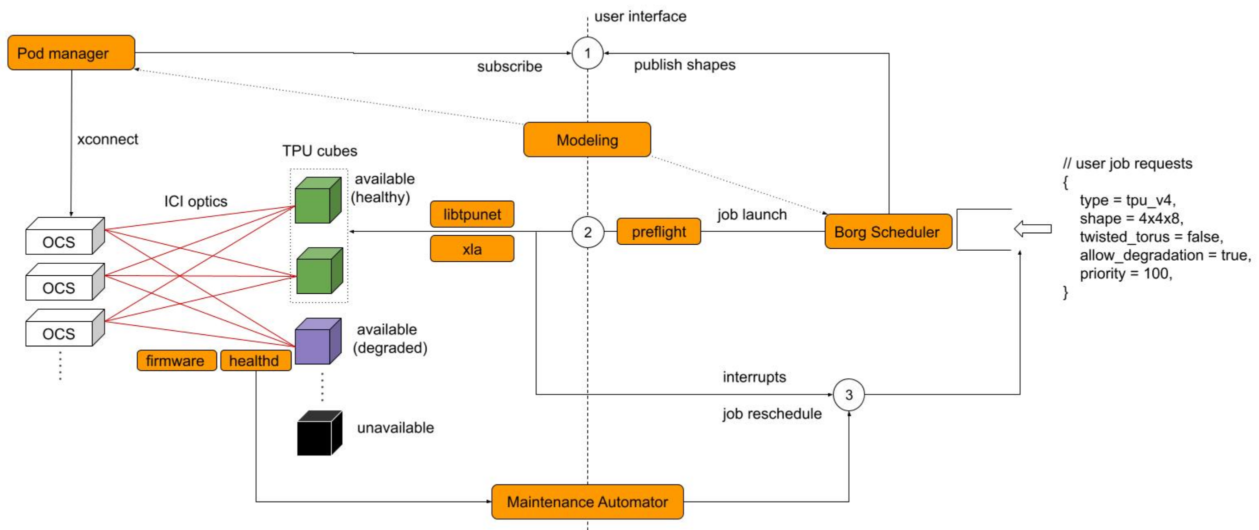
- **Routing Layer**: Packet forwarding tables are programmed by `libtpunet` with global load balancing. Each packet in a RDMA instruction goes from a source to destination TPU, indexing into the forwarding tables in each chip by the destination chip ID. The detailed routing policies are hidden from the ISA abstraction, although the `libtpunet` library can provide hints to the compiler to help guide program optimization.

- **Transaction Layer**: Compiler-generated RDMA instructions initiate hardware-mediated transfers which read data from memory and feed it to the ICI switch. A transaction spanning a group of individual RDMAs forms a collective communication operation.

Using a software-programmable ICI protocol stack allows us to flexibly cope with the complexity of a resilient 4096-node supercomputer, while allowing hardware to deal with real-time control of the links and offer high-bandwidth low-latency data transmission.

## 3 Automating Supercomputer Management

In the following we provide an overview of the end-to-end software infrastructure that we use to launch TPUv4 ML



```
// user job requests
{
    type = tpu_v4,
    shape = 4x4x8,
    twisted_torus = false,
    allow_degradation = true,
    priority = 100,
}
```

Figure 5: A TPUv4 job's life-cycle: the Pod Manager cooperates with the Borg scheduler to ask OCS to xconnect cubes, after which `healthd` preflight runs and `libtpunet` sets up the ICI network. XLA compiles programs with a distributed shared-memory system abstraction. In case a failure is detected, running jobs can be automatically interrupted and rescheduled.

training jobs, and to subsequently monitor and manage their life-cycle (see also Figure 5 for a summary).

## 3.1 Overview

When a user wishes to launch a large job[1] on a TPUv4 supercomputer, they specify their desired 3D *slice topology* in the form $(4x, 4y, 4z)$, along with other metadata. The Borg cluster scheduler [31] receives all such requests and queues them pending resource assignment. Once a job becomes eligible for scheduling, Borg will select a prospective set of cubes and then publish a xconnect request.

The Pod Manager periodically polls Borg to learn about any pending xconnect requests. For each one, it instructs the pertinent OCS switches to rotate their MEMS mirrors to establish the optical ICI physical channels. Assuming all OCS xconnects complete correctly, the Pod Manager sends a confirmation to Borg.

With Pod Manager's approval, Borg then dispatches the job binaries to the selected set of TPU machines. A preflight health check is first run to guarantee full hardware health for each TPU machine (any failures lead to Borg rescheduling onto different cubes). Following this, the ICI network is set up by `libtpunet` (i.e. validating the physical and link layers, and programming forwarding tables).

The XLA TPU compiler [3] takes the slice topology abstraction built by `libtpunet` and generates auto-parallelized TPU programs for distributed training. On each machine, the compiled TPU binary will be sent over PCIe to the TPU after which it can be executed. The above workflow is common to all ML frameworks, including TensorFlow [4], Jax [1] and Pathways [5].

During training, fleet maintenance services continuously monitor the hardware and software health of all the TPU machines. Any detected abnormality triggers a notification to Borg, which in turns notifies any affected running jobs so they can write an up-to-date model checkpoint (if possible). Once a job is rescheduled, it resumes from the latest model checkpoint. The faulty hardware is identified and sent to a repair workflow for diagnosis and repair.

The following sections describe this software infrastructure in some more detail.

## 3.2 Supercomputer Modeling

The foundation of the supercomputer software stack is a datacenter model [23] that reflects the TPUv4 machines and all related components. The model is stored in a dedicated database whose schema allows us to represent a graph of entities including racks, switches, RPC endpoints, and others. To support TPUv4 supercomputers some key entities are the TPUv4 cubes (i.e. the 16 machines with their trays and chips and static ICI inter-connection topology) as well as the ICI cabling from cubes to OCS along with additonal optics metadata. Once constructed, the model sets up the *intent* for cube deployment, job scheduling, OCS xconnect, network setup, and health checks. The model is consumed by both Borg and Pod Manager to serve as the source of truth for a each particular supercomputer configuration.

TPUv4 topologies are statically modelled up to cube size, as larger shapes require dynamic cube xconnect. For example, a single machine is a `2x2x1`, two adjacent machines can be combined to form a `2x2x2`, and so on up to `4x4x4`.

[1]We also support smaller (sub-cube) jobs. In these cases no OCS configuration is required, but the rest of the workflow is similar.

## 3.3 Cluster Scheduling

The Borg cluster scheduler [31] is responsible for assigning appropriate machines to each TPUv4 job. There are many Borg *cells* in Google's worldwide datacenters, and each cell may include several TPUv4 supercomputers. Each cell is managed by *N* replicated Borg service instances which, in combination, provide one logical Borg instance we call Borg Prime which includes a cluster scheduler.

The cluster scheduler combines the intended configuration (from the datacenter model) with its current view of the world to organize all of the TPUv4 resources it is responsible for into schedulable machine groups. Users generally select a cell in which launch their jobs, and indicate which 3D topology to use to train their model. Borg matches each user request to a set of *feasible* (usable) machines and creates a proposed assignment. In the case of multi-cube jobs, Borg publishes the proposed set of cubes to the Pod Manager and waits for it to signal xconnect success before proceeding.

Each TPU machine runs a `borglet` daemon that cooperates with Borg Prime to handle job life-cycle management. After Pod Manager approval, Borg Prime instructs each `borglet` in the assigned cubes to create a task container with the machine's TPUv4 devices[2] exposed in the task's container. The `borglet` then launches a sequence of binaries in the container, starting with the pre-flight check and finishing with the user binary.

Borg Prime and `borglet` combine to manage the response to events such as planned maintenance (e.g. firmware or software upgrades) or unexpected hardware faults. These events are aggregated from different sources, e.g. `borglet` is notified about critical local machine faults by the `healthd` daemon, and passes the details up to Borg Prime; the Pod Manager similarly forwards details about any critical OCS problems. Borg Prime also receives notification about less critical events from the Repair Automation System and the software Package Manager. In all cases, affected TPU machines are marked as unavailable, evicting any running jobs with notice, and excluding pending jobs from landing on them until things have been resolved.

Borg Prime implements priority scheduling (for higher and lower priority jobs). To help with fragmentation, Borg Prime can also choose to preempt a running workload (e.g. to relocate multiple sub-cube jobs to fit into a smaller number of cubes, or to move multi-cube workloads to a different pod so as to accommodate very large jobs). This happens in a controlled fashion, ensuring that jobs are minimally and fairly impacted.

[2]Very small jobs may use just a single TPU. In such cases, `borglet` will restrict container access to one device, and disable on-host ICI links.

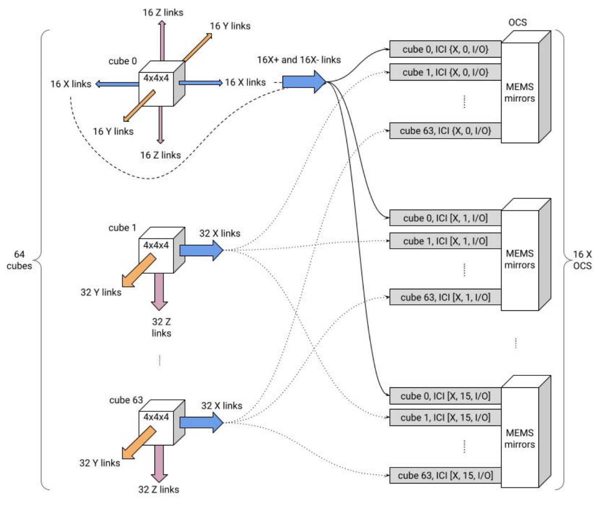


Figure 6: Each of the 64 cubes contributes two optical ICI links from two opposite sides of a ring to each of the 48 OCSes. 16 OCSes are needed for each dimension.

## 3.4 Pod Manager

Pod Manager is a highly available service critical to a TPUv4 system. It runs on dedicated network control servers that are independent of Borg, and interacts with clients such as Borg and OCS switches over the Google control plane network. The Pod Manager has two main functions: creating OCS xconnects to configure the user-requested TPU topology, and real-time monitoring of pod health .

The Pod Manager relies exclusively on model data (§3.2) to configure its services. It periodically polls the network model service for the latest information about the specific TPUv4s that it is serving, such as OCS endpoints and machines that are planned to be deployed. The OCS xconnect plan and continuous health check for every job is derived from the model. Using a model-driven Pod Manager design allows gradual deployment of a full TPUv4 supercomputer while having a subset of cubes available to customers early on.

The Pod Manager is replicated for high availability: a primary instance serves outside requests, while the remaining instances operate in hot stand-by mode so that one can quickly be elected primary if necessary. Our stand-by scheme relies on each follower continuously receiving copies of the checkpoints (also persisted externally), meaning fail-over is generally very fast. We rely on this for non-disruptive software upgrades and to tolerate hardware and software crashes.

The Pod Manager also serves as a central hub for a TPUv4 supercomputer's health monitoring. The service periodically checks the hardware health of all the optical switches by querying the OCS hardware over RPC. This telemetry is exported to Google's fleet-wide health management system (§3.6) and also used in real-time to guide fault-tolerant ICI routing optimizations (§4.2).

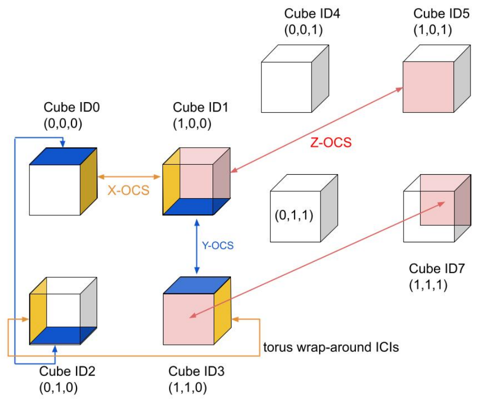


Figure 7: xconnect of an 8-cube 8x8x8 torus. Each OCS needs to pair 16 north/south OCS ports. Cube faces with the same color are interconnected to form multi-cube torus topologies.

### 3.4.1 Torus xconnect

Each 3D cube exposes 16 optical ICIs on each of the 6 faces of the $X$, $Y$, $Z$ dimensions for a total of 96 ICI links. Pod Manager assigns each of these links a unique identifier $\{cube\_id, dim, index, polarity\}$, where $cube\_id$ is a Google-wide cube UID, *index* ranges from 0–15 indicating the position within a cube face, and *polarity* can be *in* or *out*. 48 OCSes are use to xconnect these ICIs, 16 for each of the dimensions. Pod Manager gives each OCS a unique identifier $\{dim, index\}$ matching the ICI optical cables.

Fig 6 illustrates this cable connection scheme. Each OCS provides 128 ports for optical ICI connection from the cubes, allowing full connection of a single port for all 64 cubes. This scheme allows any $(4x, 4y, 4z)$ TPU topology shape to be formed, including a 4x4x4 single-cube full torus. Note that since the connected ICI and OCS have the same $\{dim, index\}$ parameters, if an OCS becomes unavailable, every cube observes one broken ICI link with the same $\{dim, index\}$ parameter.

To perform a job's cube xconnect, Pod Manager leverages its internal representation of the optical ICIs and OCSes. Fig 7 illustrates the process for a 8x8x8 torus:

- *Step 1:* Borg publishes the set of UIDs and the desired topology. Pod Manager assigns a 3D coordinate to each cube based on the topology; any cube can be chosen for any coordinate since the Pod Manager can instruct the switches to apply arbitrary port-port xconnect. For each (4x, 4y, 4z) shape, there are $x \cdot y \cdot z$ cube coordinates.

- *Step 2:* Pod Manager computes the inter-cube neighbor information based on the assigned coordinates; e.g. $(0,0,0)$ is adjacent to $(1,0,0)$ along `Xout` and `Xin`.

- *Step 3:* Pod Manager tells OCS to xconnect the ICIs $\{cubeA, dim, index, in\}$ and $\{cubeB, dim, index, out\}$ between every pair of adjacent cubes. For any topology, all 48 OCSes need to execute commands to xconnect ICI links, as all 16 ICI ports along all 3 dimensions and polarity must be connected to their remote neighbor. Each OCS can execute a variable number of commands, depending on the topology, although the OCSes for a single dimension always execute the same number. For 8x8x8 there are 8 cubes, and each OCS along the $x, y, z$ dimensions must connect 8 pairs of ports to form the torus (one per cube).

- *Step 4:* The required connections are compared against the current configuration, cached inside Pod Manager, and we filter out any connections that will remain the same. RPCs are sent to xconnect the new connections.

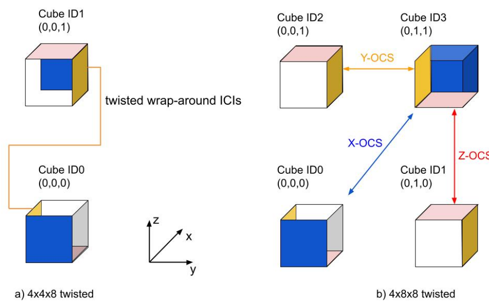


Figure 8: xconnect of (a) 2-cube 4x4x8 twisted-torus, and (b) 4-cube 4x8x8 twisted-torus. Cube faces with the same color are interconnected by OCS to form twisted-torus wrap-around ICIs. The twisted dimensions always have shorter ring sizes.

During any of the steps above, if Pod Manager determines that any port connections are infeasible, e.g., due to a hardware problem, Pod Manager will indicate this to Borg and reject the proposed set of cubes. Borg can then propose a new set of cubes set for the user's job.

### 3.4.2 Twisted-torus xconnect

In additional to a regular torus topology, we support the use of a *twisted torus* topology [7] if requested by the user. In TPUv4 twisted-torus [18], the wrap-around links are shifted with a vector offset, depending on the overall job shape. TPUv4 supports two families of twisted-torus topology: $(4k, 4k, 8k)$ and $(4k, 8k, 8k)$. Figure 8 illustrates how they are built.

For $(4k, 4k, 8k)$ shapes, the asymmetry grows along the $Z$ dimension, with the $X$ and $Y$ dimensions being identical with the same size (i.e. half the $Z$ dimension). The $X$ and $Y$ wrap-around links are shifted by a $(0, 0, 4k)$ vector offset.

For $(4k, 4k, 8k)$ shapes, the asymmetry grows along both the $Y$ and $Z$ dimensions, and the $X$ dimension has the smaller size (i.e. half of the $Y$ and $Z$ dimensions). The $X$ wrap-around links are shifted by a $(0, 4k, 4k)$ vector offset.

Cube coordinates are identical in both the regular and twisted torus case, but the latter changes which cube faces are deemed to be adjacent, and ultimately leads the Pod Manager to instruct each OCS to xconnect different north/south ports.

## 3.5 libtpunet

Once the ICI physical channels have stabilized after xconnect completion, Borg dispatches the job binaries to the host machines. The libtpunet library runs within a user's job to set up the ICI network (data and routing layer).

The first step is topology discovery. Discovery is a bottom-up process that scans each TPU's local neighbor ICI connectivity information, and runs breadth-first-search to ensure that the configured global topology matches the user request. In this process, each TPU in a job is assigned a unique chip id; this id is exposed as part of the ISA interface for RDMA instructions. The discovery process also identifies any faults that may need to be routed around, or exposed to users in the system abstraction. Topology discovery complements the intent-driven modeling of the network.

libtpunet then computes and programs the forwarding tables of each TPU based on the information curated during topology discovery. The forwarding tables are part of a job's globally optimized routing solution (more details in §4).

Along with ICI routing programming, libtpunet sets up the link-level flow-control buffer size, in proportion to the the link RTT. libtpunet also programs the configuration of consistent clocking on a job's distributed TPU set. The clock configuration is generated using a minimum spanning tree, factoring in the longest RTT of any ICI path. Using a consistent clock enables precise timestamps for performance tracing and debugging.

Finally, libtpunet starts an *ICI session*, allowing the use of various compiler-generated collective ops with RDMA. This is done by synchronously enabling the data layer of each ICI link across its two ends. An ICI handshake is performed in hardware to confirm the reliable data link enable request is initiated from both ends of the link.

libtpunet stays active through the job's lifetime to monitor the health of the ICI session. If any TPU observes an error, the link layer comes down, or the driver panics, a PCIe MSI-X interrupt is raised to libtpunet, which notifies Borg to initiate rescheduling.

## 3.6 Hardware Maintenance and Recovery

At global fleet scale, disruptive maintenance events (e.g. hardware repair or replacement, or critical software/firmware upgrades) occur relatively frequently. To maximize overall maintenance efficiency, Google operates a fleet automation system. Its remit covers hardware failure diagnosis, a hardware recovery workflow, and system software package installation (e.g. host kernel or device firmware).

Events generated by the fleet maintenance automation system send notifications to Borg to evict running jobs on impacted machines; any evicted jobs are queued for rescheduling. In case of suspected failures, the impacted hardware is sent to a repair workflow that marries automatic diagnosis with technician input if needed. Once the hardware is recovered it flows through an automated QA process before rejoining the resource pool. For our TPUv4 supercomputer we extended this system with continuous TPU hardware health telemetry, explicit preflight checks before job launch and a scheme for on-line ICI link repair.

### 3.6.1 healthd

We added a healthd daemon on every TPUv4 machine to perform real-time monitoring of hardware parts including the 24 unidirectional ICI links, the PCIe channels between the TPUs and CPUs, and the 4 TPU ASICs themselves. A set of hardware symptoms are defined for each of these components based on the telemetry data gathered by healthd. healthd consumes the same model as Pod Manager which provides the necessary details about monitoring endpoint, firmware, and ICI cable metadata.

For each ICI link, the cable connection and associated link quality are continuously checked against the modeled values and a set of predefined thresholds. Any detected *symptoms* are ranked by their criticality, with severe symptoms leading healthd to notify Borg to evict and reschedule affected jobs.

### 3.6.2 Preflight Check

A preflight check runs before every user job to ensure hardware is healthy. We currently include two different checkers: an *end-to-end* check validates the TPU hardware by running a mini sample workload, while an *intent-driven* checker validates physical-level hardware metrics against a set of golden "within spec" thresholds. The former provides broad coverage of both the hardware and software components including the TPU driver, firmware and libtpunet which all interact with the underlying chip and ICI; the latter allows detection of less obvious issues such as substandard link quality metrics. If the preflight checks fail, borglet will indicate to Borg Prime that the job should be rescheduled.

### 3.6.3 Online ICI Link Repair

For TPUv4, ICI link repair can be carried out online, automatically coordinated across the two ends of a link so that recovery can be reliably verified. The two endpoints can span two different machines, or a machine and an OCS switch. The Pod Manager coordinates all ICI network maintenance through *ICI link drains*. A drained ICI link is automatically excluded from user applications, although the TPU compute resources are not impacted (i.e. jobs can still land on the TPU machines providing they do not use the broken ICIs).

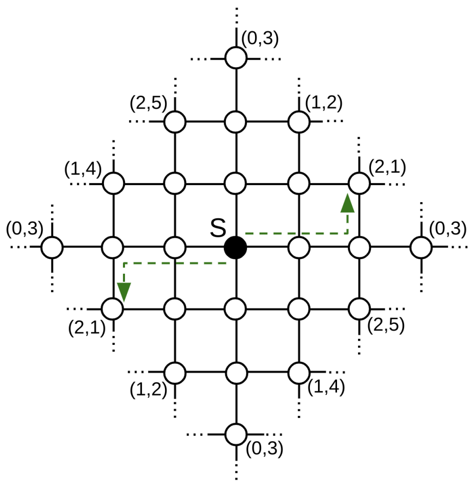


Figure 9: Shortest-path routes from the origin (S) in a 2D `3x6` twisted torus are confined to a diamond-shaped region [7]. Destinations along the boundary are labeled with their coordinates, with each destination appearing at least twice, meaning we need tiebreaking to pick among the shortest paths. For example, there are two possible paths to $(x = 2, y = 1)$, shown as dashed lines, and there are four possible paths to $(0,3)$.

## 4 ICI Routing

We use multi-hop packet routing over high-bandwidth ICI links to provide fast TPU RDMA and collectives. ICI routing allows RDMA packets to be sent between arbitrary pairs of TPUs in the pod, and can work around certain ICI faults. The ICI forwarding tables are programmed once by `libtpunet` at job start-up, and remain fixed over the job's lifetime. Each source-destination pair sends packets along a single predetermined path through the ICI topology.

While simple, this approach is sufficient to achieve high performance on the typical collective communication patterns (e.g. all-gather, reduce-scatter, all-reduce, all-to-all) that arise during parallel decomposition of ML models [33].

There are two cases described in this section where `libtpunet` must carefully select a single path among multiple candidates to satisfy the ICI forwarding table constraints: *tiebreaking* and fault-tolerant *wild-first routing*. We perform path selection off-line using an integer linear programming approach and the results of this optimization are cached. This allows `libtpunet` to quickly load the precomputed solution during ICI network setup.

### 4.1 Fault-free Routing

When configured for regular torus topology, ICI uses dimension-order routing (DOR) [11]: all packets route one dimension at a time in a fixed order (e.g. $X$ then $Y$ then $Z$) following a shortest-path from source to destination in the torus. The dimension order is chosen so that longer dimensions of the torus are routed first as described in [8]. DOR is sufficient to balance load for the common traffic patterns of ML jobs. It can also be made deadlock-free with just two virtual channels [10], making it inexpensive to implement.

One complication occurs when a packet needs to travel exactly halfway around a dimension since tn this case, there are two shortest paths to choose from. For example, in an `8x8x8` torus, routing a packet from source $(x = 1, y = 0, z = 0)$ to destination $(x = 5, y = 0, z = 0)$ can travel 4 hops in either the $X^+$ or $X^-$ direction.

We handle this *tiebreak* case algorithmically: a packet takes the positive direction when the relevant source node coordinate along that dimension is even, otherwise uses the negative direction. In our example, the source $(x = 1, y = 0, z = 0)$ has an odd $X$ coordinate, so tiebreaking chooses $X^-$. This scheme balances load for common all-to-all traffic patterns.

Routing in the twisted torus also uses DOR and is deadlock free with two virtual channels. However tiebreaking in the twisted torus is more complicated because the dimensions are not separable as in the regular torus: Figure 9 illustrates this using two dimensions for simplicity. We decided to fold handling tiebreaking in the twisted torus case into a more general integer-linear programming framework that also handles fault-tolerant routing (§4.3). This obviates the need to develop an explicit tiebreaking algorithm in this case.

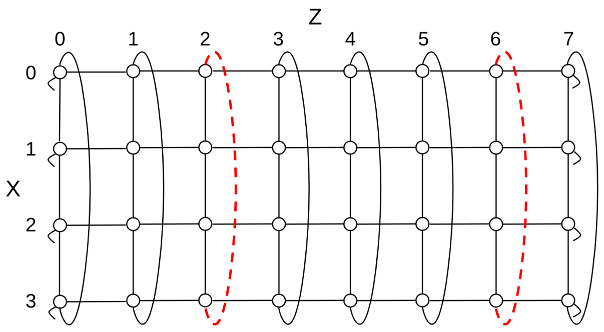


Figure 10: Example of ICI links impacted by an OCS being unavailable along the $X$ dimension of a `4x4x8` torus. The unavailable OCS results in two unavailable $X$ links along one $XZ$ plane of the torus; the other $XZ$ planes are unaffected. Unavailable links are emphasized with dashed red lines. The connectivity of the OCSs creates a periodic fault pattern, where unavailable links repeat every 4 hops along $Z$. This pattern is due to the OCS connectivity (Figure 6).

### 4.2 Fault-tolerant Routing

The reconfigurable ICI architecture is inherently resilient to machine outages due to the ability for dynamic cube selection. In `libtpunet`, further resilience against OCS unavailability events is added by supporting *fault-tolerant routing*. If an OCS is unavailable, a sparse set of links becomes unavailable. As shown in Figure 10, the patterns of unavailable links are

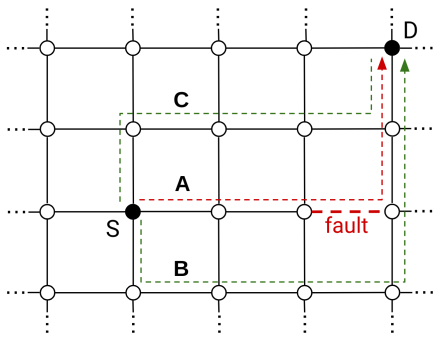


Figure 11: Example routes (dashed paths) between a source S and destination D. The $XY$ dimension order route (A) crosses the unavailable link (red dashed line) so cannot be used. The two possible $yXY$ wild-first routes (B and C) take a single hop in $Y$ before continuing with $XY$ DOR, avoiding the fault.

highly symmetric with respect to the cube granularity as a consequence of cube-OCS connectivity. With a small amount of routing flexibility, packets can avoid the broken links. We optimize routing algorithms for these scenarios off-line.

The path between a source-destination pair selected by dimension-order routing (DOR) becomes unusable if it crosses an unavailable link. In this case, alternative paths are created using *wild-first routing* (WFR). In WFR, a packet is allowed to take at most one wild hop along each dimension before reverting to use DOR to its destination.

Fig 11 shows an example of WFR routing in a two-dimensional torus. In this example, the dimension order is $X$ then $Y$ and only wild hops along $Y$ (either $Y^+$ or $Y^-$) can help avoid unavailable links. We use the shorthand $yXY$ to denote a wild-first routing algorithm that takes a wild hop along $Y$ before continuing with $XY$ dimension-order route. The $yXY$ algorithm can avoid any single unavailable link in the $X$ dimension.

There is a *sandwich rule* that captures the fault tolerance of WFR: to avoid a fault in one dimension, hops in another dimension must occur before and after it in the routing algorithm. For the $yXY$, $X$ is "sandwiched" by hops in $Y$, so it avoids faults in $X$. Similarly, the $xYX$ algorithm can avoid faults in Y. Extending to three dimensions, the $xyZYX$ algorithm can avoid a single fault in both the $Y$ and $Z$ dimensions.

The development of WFR was influenced by the microarchitecture of the ICI switches. While beyond the scope of this paper, WFR can be made deadlock-free with two virtual channels with one restriction: the wild hop order must be the reverse of the dimension order. For example, $xyZYX$ is deadlock-free with two virtual channels, but $yxZYX$ is not.

## 4.3 Offline Route Optimization

The previous sections described situations where multiple candidate paths can be produced, either due to tiebreaking or due to the wild-first routing algorithm. The ICI switch implementation, however, makes use of static forwarding tables, which are programmed with a single path for each source-destination pair. As mentioned previously, we formulate path-selection as an integer-linear program (ILP), calculating solutions offline and caching them for use at runtime.

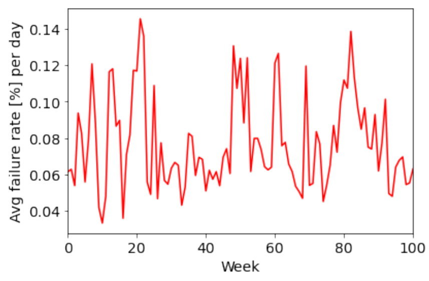
The goal of the ILP is to maximize the throughput of a predefined traffic pattern by solving a maximum concurrent flow problem [27]. All-to-all is typically chosen as the traffic pattern and supplemental constraints ensure other collectives (e.g. all-reduce) perform well. Per-path variables are constrained to Boolean values to adhere to the static routing constraints, with exactly one path per source-destination pair.

In both the fault-free and fault-tolerant cases, the ILP is designed to exploit translational symmetry to reduce the number of variables [30]. This makes finding optimal solutions for practical network sizes tractable. The torus and twisted torus are both vertex symmetric, so a single set of path variables can be used for a canonical source and then translated to all other sources. When an OCS is unavailable, the fault pattern is periodic with cube granularity, as was shown in Figure 10. While the resulting topology is less symmetric than the fault-free cases, the ILP can be still restricted to a set of canonical sources. This also enables a single canonical case to be solved offline and cached. The canonical fault pattern, trivially translated from the actual fault pattern, is initialized during network setup.

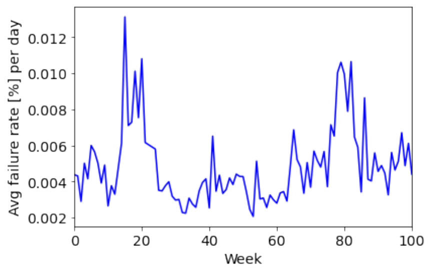
| Slice | No faults (GB/s) | 1 fault (GB/s) | 1 fault (% of no faults) |
|---|---|---|---|
| 4x4x4 | 75.9 | 70.0 | 92.2% |
| 4x4x8 twisted | 62.1 | 63.2 | 101.7% |
| 4x8x8 twisted | 54.3 | 53.7 | 98.8% |

Table 2: Measured all-to-all throughput with (a) all OCSs healthy and (b) one OCS unavailable. The last column shows the throughput of the single-fault case versus the healthy case.

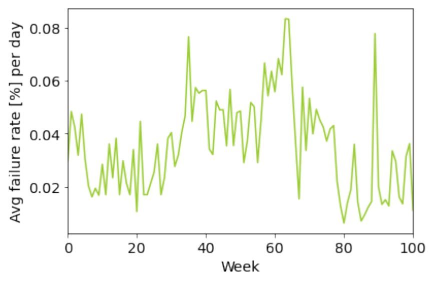
Table 2 compares measured all-to-all throughput in fault-free and single OCS failure scenarios when the forwarding tables are optimized using this methodology. For the regular torus networks, a single unavailable OCS reduces the ideal all-to-all performance to $15/16 \approx 93.4\%$ of the fault-free network. This corresponds to losing one of the 16 links along one face of the 4x4x4 cubes that connect to the OCSs. Interestingly the twisted torus shows better resilience, with the 4x4x8 performance *improving* slightly in the presence of an unavailable OCS . This is because of the flexibility offered by tiebreaking: by balancing different tiebreaking paths, traffic can be shifted from one dimension to another. This balancing is not possible in the regular torus. The slight improvement in the 4x4x8 case is a good illustration that the ILP formulation is only a proxy for real world performance.



(a) Machine daily failure rate.



(b) ICI link daily failure rate.



(c) OCS daily failure rate.

Figure 12: Weekly statistics of a supercomputer's hardware failure and recovery, including TPU machines, ICI cables, and OCS.

## 5 Fleet Statistics

In this section, we describe Google's fleet experience of operating TPUv4 supercomputers over the past two years. We focus on the software stack's automatic management of OCS xconnect, faults, and overall system availability.

### 5.1 Cube Reconfigurations

Thousands of training jobs are submitted to Google's TPUv4 supercomputers every day. Figure 13 shows a sample supercomputer's OCS xconnect actions over two months, correlated with the pod's number of admitted jobs. A higher number of jobs normally incurs more OCS xconnect changes because the Pod Manager updates ICI port xconnect for each job on arrival. We also see OCS xconnect changes when there are very large and/or long running training jobs which will experience reschedules to handle maintenance and failures. Overall, TPUv4 supercomputers function reliably with many tens of thousands of OCS xconnect changes per pod per day.

Training jobs can vary drastically in size and system topology, ranging from sub-cube mesh jobs for small scale experiments to large jobs that use almost the entire pod for LLM pretraining. We anecdotally observe that many embedding-heavy recommendation models adopt the twisted-torus topology, and some transformer-based models use model parallelism across more irregular torus topologies.

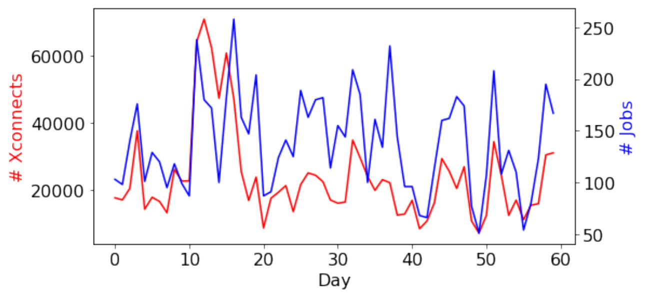


Figure 13: The OCS xconnect actions and jobs admitted by a TPUv4 supercomputer over two sample months.

### 5.2 Hardware Maintenance Automation

TPUv4's reconfigurability and fault-tolerant routing allow for resiliency against machine and OCS outages. Figure 12 shows the average failure rate of different hardware components in each supercomputer.

Faults are diagnosed at the TPU machine, ICI link, and OCS levels. The Pod Manager and `healthd` automate the repair and recovery process. In an average supercompuer, each day, 0.08% of the TPU machines, 0.005% of the ICI cables, and 0.04% of the OCS experience a failure. While these values are small, the number of jobs that are impacted by hardware outages is non-trivial because each supercomputer has a large number of machines, ICIs, and OCS. Machine and ICI outages are automatically tolerated by reconfiguring jobs to use spare healthy cubes. An OCS outage has larger blast radius as it can impact all cubes in a supercomputer. Fault-tolerant ICI routing lets us tolerate OCS outages with some performance impact; and we priority recovery time for OCS components compared with others to minimize this.

### 5.3 Fault-tolerant Jobs

In our experience to date, 95% of all TPUv4 training jobs opt in to fault-tolerant ICI routing so they can be resilient to OCS outages; the remaining jobs opt-out to rule out performance non-determinism caused by different routing strategies. Figure 14 shows the ratio of all fleet-wide jobs actively running with fault-tolerant routing across a 8-month sample period. In general at any time, fewer than 2% of the jobs are running with fault-tolerant routing. This quantity is highly correlated with OCS maintenance events and the per event recovery time. The spike around day 60 is due to a planned fleet-wide upgrade of OCS parts to improve reliability.

Fault-tolerant ICI routing comes with a performance penalty due to more congested traffic around faulty links. The load imbalance affects collective operations including all-to-all and all-reduce. We measured the performance impact across a range of key Google workloads spanning *Recommendation Models* (RM), *Large Language Models* (LLM), and BERT-based [13] models. Table 3 summarizes our results.

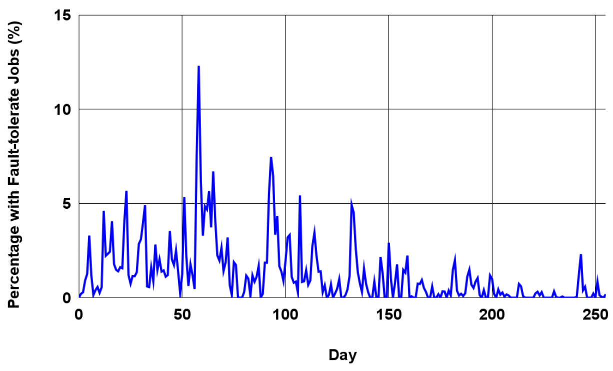


Figure 14: Percentage of jobs using fault-tolerant routing (OCS outage resiliency) over an 8-month window.

For all-to-all heavy workloads, the step time degradation is not significant because the offline routing optimizer has minimized, if not improved, all-to-all performance. This is particularly true for embedding-heavy twisted-torus shapes. The all-reduce workloads experience higher performance impact because the nearest-neighbor communication pattern receives 50% throughput hit. The impact on all-reduce operation can be improved by smarter overlapping between compute and communication. Overall, all workloads experience a small slowdown in step time.

## 6 Related Work

The architecture decision to use OCS for TPUv4 is discussed in [18], while [20] and [19] evaluate the design of prior generation static TPU pods. This paper describes the software ecosystem for TPUv4 and how it achieves resilience at scale. The usage of OCS in production-scale data center networks was described in [25], discussing considerations for scalability, cost, and topology engineering. This paper focuses on OCS use TPUv4 supercomputers.

Previous work has covered circuit switching for supercomputers [28] and proposed topology engineering for ML training [32]. Nvidia uses a 2-tier NVswitch-based fat tree network over NVlink for inter-GPU collectives. These represent a different design point compared with ours: OCS simplifies the ICI network design compared to introducing packet switches because it establishes dedicated physical channels without the need to control shared traffic, while the lower purchasing price and stand-by power also reduces operating cost [20].

Twisted tori are due to [6, 26]; the specific $(4k, 4k, 8k)$ and $(4k, 8k, 8k)$ twisted torus shapes supported TPUv4 conform to those of [7]. Finally software-defined datacenter networks have been described extensively in the literature (e.g. [14, 16, 25, 29]). To our knowledge, we are the first to describe this approach for an exascale supercomputer.

| Workload | Topology | Step Time Slowdown |
|---|---|---|
| RM-1 | 4x4x8 twisted | 0.5% |
| RM-2 | 4x4x8 twisted | 3% |
| RM-3 | 4x4x8 twisted | 3.9% |
| RM-4 | 4x4x8 twisted | 8.6% |
| RM-5 | 4x4x8 twisted | 8.3% |
| RM-6 | 4x4x8 twisted | 4.7% |
| LLM-1 | 4x4x8 | 2.6% |
| BERT-1 | 4x4x4 | 1.2% |
| BERT-2 | 4x8x8 | 3.2% |

Table 3: Performance impact from fault-tolerant ICI routing.

## 7 Future Work

Our main short term priorities are improving the performance and recovery overhead of TPU pods: ML supercomputing hardware is in high demand, and every little helps. In future, as well as supporting increased line rate for ICI links, we plan to introduce a randomized routing capability to ICI switches to enables better load balancing for both torus and twisted torus topologies in the presence of faults, particularly for nearest-neighbor communication patterns.

We also plan tighter integration between OCS-based configurability and workload reconfiguration by allowing jobs to continue mostly unaffected by failures. Our approach here is to provision a hot-standby cube in response to an outage event, and directly migrate accelerator state to the new TPUs without ever writing a persistent checkpoint. This work involves changes to the Borg scheduler (to provision on demand), `libtpunet` (to dynamically adjust a built ICI session) and the Pathways ML runtime [5] (to manage state transfer).

## 8 Conclusion

The TPUv4 supercomputer is an exascale 4096-chip computing system that addresses the availability and scalability challenges of fast-paced ML model evolution. TPUv4 offers approximately 2.1× performance compared to the previous generation, but also features cube-level *reconfigurability* based on optical circuit switching and uses fault-tolerant ICI routing to allow operation if switches fail.

This paper has described the end-to-end software infrastructure for TPUv4 that provides flexibility for topology, routing, scheduling, interrupting, monitoring, and hardware health management. TPUv4's software-defined ICI networking approach enables strong fault resiliency to machine and switch outages at scale. The software has been operating in production since 2020, running TPUv4 supercomputers for both Google Cloud clusters and internal users, and sustaining 99.98% system availability,

## 9 Acknowledgements

We wish to thank our collaborators: Xiangyu Dong, Sebastian Mugazambi, Sameer Kumar, Andy Koch, Jianqiao Liu for the project inception, and Wenbo Zhao, Hongji Li, Rajendra Gottipati, Yuechao Pan, Justin Duan, Peter Gavin for their work on continuous measurement and monitoring. We also thank Kais Belgaied and Varinder Singh for their support during the design and development of our software infrastructure. Finally, special thanks go to Robert Hundt, Norm Jouppi, Aamer Mahmood, and the anonymous NSDI reviewers for their feedback and suggestions on the paper.

## References

[1] JAX: Autograd and XLA. https://github.com/google/jax [Accessed: 2023-04-26].

[2] The TPUv4 Exaflop ML Supercomputer. https://cloud.google.com/blog/topics/systems/tpu-v4-enables-performance-energy-and-co2e-efficiency-gains [Accessed: 2023-04-26].

[3] The XLA TPU Compiler. https://github.com/openxla/xla [Accessed: 2023-04-26].

[4] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[5] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. Pathways: Asynchronous Distributed Dataflow for ML, 2022.

[6] Wendell J Bouknight, Stewart A Denenberg, David E McIntyre, JM Randall, Amed H Sameh, and Daniel L Slotnick. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369–388, 1972.

[7] Jose M Camara, Miquel Moreto, Enrique Vallejo, Ramon Beivide, Jose Miguel-Alonso, Carmen Martínez, and Javier Navaridas. Twisted torus topologies for enhanced interconnection networks. *IEEE Transactions on Parallel and Distributed Systems*, 21(12):1765–1778, 2010.

[8] Dong Chen, Noel A. Eisley, Philip Heidelberger, Robert M. Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L. Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2011.

[9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[10] W. J. Dally and C. L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. Comput.*, 36(5):547–553, may 1987.

[11] William James Dally and Brian Patrick Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019.

[14] Andrew D Ferguson, Steve D Gribble, Chi-Yao Hong, Charles Edwin Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. Orion: Google's Software-Defined Networking Control Plane. In *NSDI*, pages 83–98, 2021.

[15] Saurabh Gupta, Tirthak Patel, Christian Engelmann, and Devesh Tiwari. Failures in large scale systems: long-term measurement, analysis, and implications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[16] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, et al. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 74–87, 2018.

[17] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan

Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[18] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th annual international symposium on computer architecture*, 2023.

[19] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Communications of the ACM*, 63(7):67–78, 2020.

[20] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, 2017.

[21] Sameer Kumar, James Bradbury, Cliff Young, Yu Emma Wang, Anselm Levskaya, Blake Hechtman, Dehao Chen, HyoukJoong Lee, Mehmet Deveci, Naveen Kumar, Pankaj Kanwar, Shibo Wang, Skye Wanderman-Milne, Steve Lacy, Tao Wang, Tayo Oguntebi, Yazhou Zu, Yuanzhong Xu, and Andy Swing. Exploring the limits of Concurrency in ML Training on Google TPUs, 2021.

[22] Sameer Kumar and Norm Jouppi. Highly Available Data Parallel ML training on Mesh Networks, 2020.

[23] Jeffrey C. Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with Modeling Network Topologies at Multiple Levels of Abstraction. In *17th Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[24] OpenAI. GPT-4 Technical Report, 2023.

[25] 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. Jupiter evolving: Transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of ACM SIGCOMM 2022*, 2022.

[26] Carlo H Sequin. Doubly twisted torus networks for VLSI processor arrays. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 471–480, 1981.

[27] Farhad Shahrokhi and D. W. Matula. The Maximum Concurrent Flow Problem. *J. ACM*, 37(2):318–334, apr 1990.

[28] John Shalf, Shoaib Kamil, Leonid Oliker, and David Skinner. Analyzing ultra-scale application communication requirements for a reconfigurable hybrid interconnect. In *SC'05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 17–17. IEEE, 2005.

[29] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in Google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[30] Brian Towles, William J. Dally, and Stephen Boyd. Throughput-Centric Routing Algorithm Design. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, page 200–209, New York, NY, USA, 2003. Association for Computing Machinery.

[31] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[32] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing Network Topology and Parallelization Strategy for Distributed Training Jobs, 2022.

[33] Yuanzhong Xu, HyoukJoong Lee, Dehao Chen, Blake Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, Dmitry Lepikhin, Andy Ly, Marcello Maggioni, et al. GSPMD: general and scalable parallelization for ML computation graphs. *arXiv preprint arXiv:2105.04663*, 2021.